図解　16ビットマイクロコンピュータ

8O286の使い方

本岡善剛著

オーム社

図解　16ビットマイクロコンピュータ

80286の使い方

本岡善剛 著

オーム社

UNIX は AT & T 社ベル研究所の商標．

CP/M-86 は米国ディジタルリサーチ社の商標．

MS-DOS，XENIX は米国マイクロソフト社の商標．

I^2ICE は米国インテル社の商標．

はしがき

初期のマイクロプロセッサは，その機能を限定することによって半導体集積回路にコンピュータのCPU（中央処理装置）を実現したものであった．したがって，その大きさにおいても能力においても小さなコンピュータを意味した．しかし，半導体技術の発達によって，現在のマイクロプロセッサは決して小さなコンピュータではなく，その能力は10年ほど前の大型汎用コンピュータと同レベルかそれ以上のものになっている．

しかし，マイクロプロセッサの重要な使命は特別な権利をもつ技術者に限らず，だれに対しても広くそのコンピュータの能力を提供することである．このことはまたマイクロプロセッサの大きな魅力でもある．このためマイクロプロセッサのアーキテクチャは，ミニコン，大型汎用コンピュータの技術を学ぶ一方で，それをだれもが利用できるような形に昇華されてきた．この点においてインテル社の8086は最も成功したマイクロプロセッサの1つであろう．

本書で述べる80286は，8086の上位に当たるマイクロプロセッサであるが，この2つのマイクロプロセッサの間には単なる改良型というもの以上の大きな違いがある．このためインテル社は，80286以後のマイクロプロセッサを8086などに対して第3世代のマイクロプロセッサと呼んでいる．

80286はマルチタスクオペレーティングシステムを実行することを念頭において設計されたマイクロプロセッサである．この意味で80286は16ビットのマイクロプロセッサであるが，32ビットのマイクロプロセッサ80386と同レベルに位置する．80286は，マルチタスクオペレーティングシステムを設計するうえで重要な処理をマイクロプロセッサのハードウェアとマイクロコードで処理する．

80286は高速の8086としても利用できるが，本書ではマルチタスクオペレーティングシステムを設計するうえで重要な80286の基本的機能と命令を中心にまとめた．8086と共通な命令の詳細な解説は他の8086関係のすぐれた各著書にまかせることにするが，本書自体は8086を知らなくても十分に読めるように解説

した．

本書を執筆するに当たっては多くの方々のお世話になった．これらの方々の協力なしには本書は完成しなかった．この場を借りて心から謝意を表します．インテルジャパン株式会社トレーニングセンター前マネージャー大野民生氏は，本書の執筆を私に勧めてくださった．同社のマーコムマネージャー松本芳夫氏はインテル社のユーザーズガイド，データシートを利用する上で協力的に対応していただいた．また，同社トレーニングセンターのスタッフ全員からは，執筆において絶え間のない励ましをいただいた．特に池田聡氏は私の読みにくい原稿をていねいに読み，多くの注意点を指摘していただいた．

本書にあげたプログラムはすべてインテル社 MDS シリーズⅣと I²ICE また SIM 286 を使用して動作を確認したものであり，その他の内容についても正確さには細心の注意を払った．しかし，なお記述中に誤りが残っているとすれば，それはすべて著者の責任である．

最後に，技術的知識とそれらを本にまとめるセンスとは別のものであり，私の原稿がこのような本にまとめることができたことはオーム社出版部の方々の努力に負うところが大である．心から感謝の意を表します．

1987 年 3 月 12 日

著者しるす

目　　次

4. 特権レベルによる保護

5. 割り込み処理

6. タスクとタスクスイッチ

7. 保　護　例　外

8. 80286 のハードウェア

9. 数値演算コプロセッサ 80287

10. プログラム開発

付　録

1. 80286の概要

マイクロプロセッサ 80286 は，16 ビットのレジスタ群と 16 ビットのデータバスをもつプロセッサで，マイクロプロセッサ 8086 の応用プログラムを，ほとんど書き換えることなく実行することができるものである．80286 は，2 つの動作モードをもつ．**リアルモード**と呼ばれるモードでは，80286 はスピードの速い 8086 として動作する．しかし，他方の**プロテクトモード**と呼ばれるモードにおいて，80286 は本来の能力を発揮する．

1-1　8086から80286へ

〔1〕 **16ビットマイクロプロセッサの歴史**　インテル社のマイクロプロセッサの歴史は表1·1に示すように，4ビットの4004において始まった．4004の目的は電卓の制御だったが，メモリに記録したプログラムを順次実行するストアードプログラム方式，または演算機能をもつことなどは，4004をコンピュータの中央処理装置（CPU）と呼ぶに十分な特徴であった．

8ビットの8080に至って，CP/Mなどのディスクオペレーティングシステム（DOS）も作られ，マイクロコンピュータの開発システム，またはパーソナルコンピュータにも使用された．しかし，8085はどちらかといえば，制御に使用されることが多く，かつての16ビットミニコンピュータと比較して，性能において大きな開きがあった．

しかし，16ビットの8086においては，部分的にはミニコンピュータの置き換えに利用されるまでの性能をもつに至った．8086では，CP/M-86，MS-DOSなどのDOSが普及し，8086上で走る応用プログラムは膨大な量になる．世界におけるその総額は1986年において60億ドルといわれている．

〔2〕 **80286の必要性**　図1·1(a)に示すようなシングルタスク，シングルユーザのシステムでは，8086は十分な能力をもつが，同図(b)，(c)に示すような**マルチタスク**のシステムを8086を使用して実現することが困難な場合がある．ここで，タスクとは実行可能なプログラムのことである．マルチタスクシステムは，メモリに複数のプログラムを置き，図1·2に示すように，必要に応じてCPUの使用権を他のタスクに渡しながら処理を進めることができる．

マルチタスクシステムの応用としては，図1·1(b)に示す**リアルタイムシステム**を作ることができる．このようなシステムでは，ロボットがベルトコンベアに部品を置く作業をしているが，環境の変化に応じてリアルタイムに（すぐに），次のプログラムを実行することができる．2つのタスクはまったく別のプログラムでもよいし，また，同じプログラムでもかまわない．

図1·1(c)に示すシステムでは，1つのコンピュータに2台以上の端末が接続され，複数のユーザがそれぞれの端末から1つのシステムを共有することができる．2人のユーザは別々のプログラムを実行してもよいし，また，同じエディタ

表 1・1 インテル社のマイクロプロセッサの発展

発表時期	マイクロプロセッサ	処理データの大きさ	実メモリ空間	I/O 空間
1971	4004	4 ビット	4K バイト(ROM) 5 120 ビット(RPM)	16×4 ビット(入力ポート) 16×4 ビット(出力ポート)
1972	8008	8 ビット	16K バイト	
1974	8080	8 ビット	64K バイト	256 バイト
1976	8085	8 ビット	64K バイト	256 バイト
1978	8086	16 ビット	1M バイト	64K バイト
1982	80186	16 ビット	1M バイト	64K バイト
1982	80286	16 ビット	16M バイト	64K バイト
1985	80386	32 ビット	4G バイト	64K バイト

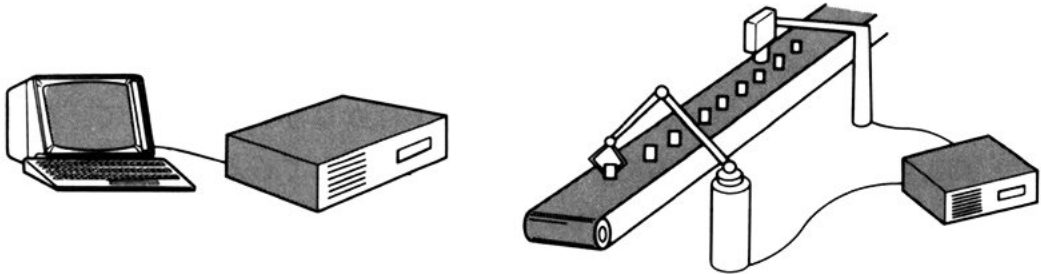
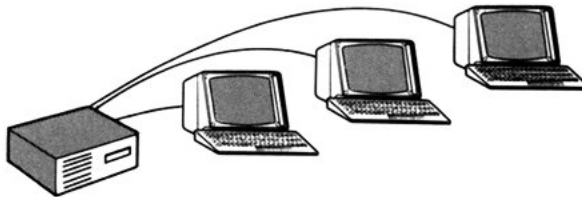
1K バイトは 1024 バイト,
1M バイトは 1024K バイト,
1G バイトは 1024M バイトをそれぞれ表す.

(a) シングルタスク, シングルユーザシステム　　(b) リアルタイムシステム

(c) マルチタスク, マルチユーザシステム

図 1・1 コンピュータシステムの利用形態

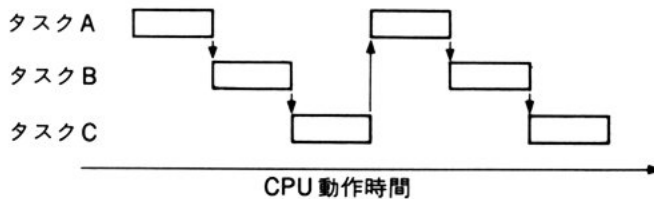



図 1・2 CPU 時間の分割使用

（編集プログラム）を使用してユーザ独自のテキストを作成することもできる．このシステムでは，各ユーザが実行するプログラムはそれぞれタスクとなり，各タスクがCPUを**時分割**（**タイムスライシング**）で使用する．もちろん，CPUを時分割で使用することによって，図1·2に示すように，1つのタスクの実行時間は長くなる．しかし，ユーザが端末を使用して作業するとき，このような実行時間の変化はほとんど感じないだろう．各ユーザは他のユーザとは関わりなく，自分だけがシステムを独占しているように使用することができる．

ここで，マルチタスクシステムでは，タスクへのCPUの切り換え，メモリ管理，ファイル管理などの作業が必要である．たとえば，図1·1(c)のシステムにおいて，あるユーザが実行させたプログラムが暴走することによって，別のユーザのプログラムの実行をつぶすようなことがあってはならないし，ディスクに格納した大切なデータを他のユーザに読まれたり，書き換えられるようなことがあってはならない．これらはオペレーティングシステム（OS）の仕事であるが，CPUの能力によっては，OSのオーバヘッドばかりが大きくなって，本来のユーザプログラムに分け与えるCPU時間が少なくなってしまう．こうなると，ユーザはCPUを時分割で共有していることを感じざるをえず，実用にならない場合がある．

80286は，マルチタスクシステムに必要な，タスクスイッチ，メモリ管理，各種の保護機能をCPU自体がもち，8086の応用プログラムをマルチタスクで実行するためのマイクロプロセッサである．

1-2 マイクロコンピュータの構成

〔1〕 **マイクロコンピュータの基本構成**　　マイクロコンピュータは，基本的には図1·3に示すように，**CPU**，**メモリ**，**I/O**（Input/Output，入出力装置）の3つの要素によって構成される．CPUは，演算などのコンピュータにおいて最も中心となる仕事をする部分である．また，その他にコンピュータ全体を能動的に制御するいくつかの機能をもつ．メモリは，プログラム，データを一時的に記録する装置で，CPUによってリード（メモリからCPU内部のレジスタへデータを読む動作），ライト（CPU内部のレジスタからメモリへデータを書く動作）が実行される．I/Oには周辺装置との**インタフェース**を接続する．CPUとI/Oインタフェース間のデータのリード，ライトは，メモリと同様の方式で実行される．インタフェースは，データを外部の周辺装置に適合するように信号を変換する．

〔2〕 **バ　　ス**　　これらの3つの要素はバスと呼ばれる伝送路によって接続されている．バスは，**制御バス**，**アドレスバス**，**データバス**に分類される．制御バスは，「メモリからデータをリードする」，「メモリにデータをライトする」，「I/Oからデータをリードする」，「I/Oにデータをライトする」などの**コマンド信号**をCPUからメモリ，またはI/Oに送るために使用する．

たとえば，「メモリからデータをリードする」ことを表すコマンド信号をCPU

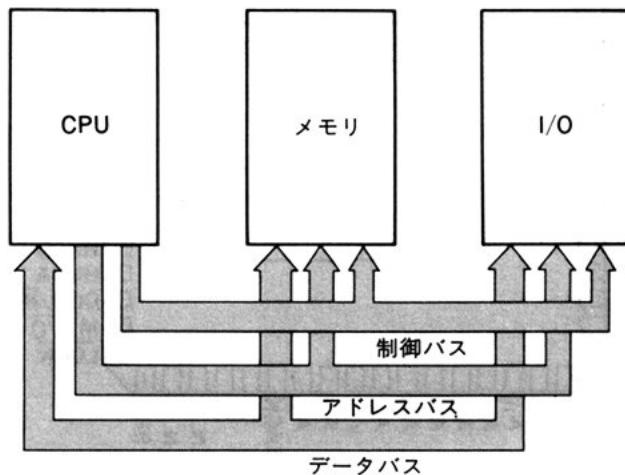



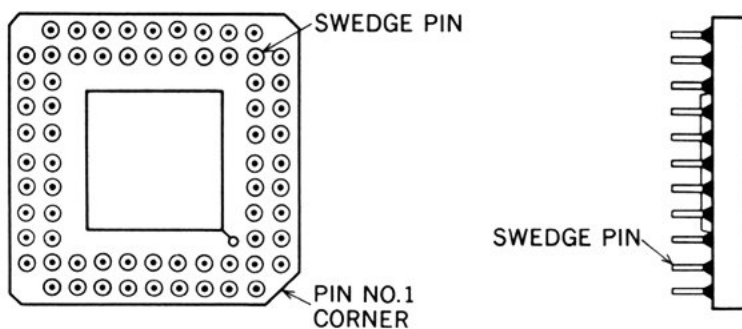
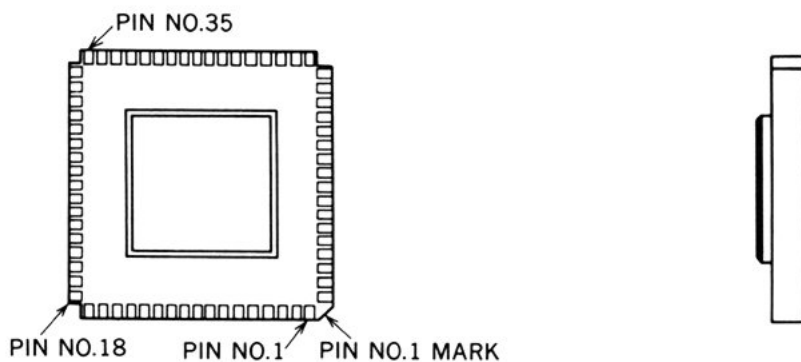
図 1·3　マイクロコンピュータの構成

(a) PGA(ピングリッドアレイ)パッケージ

(b) JEDECタイプLCC(リードレスチップキャリア)パッケージ

図 1・4 80286 マイクロプロセッサ

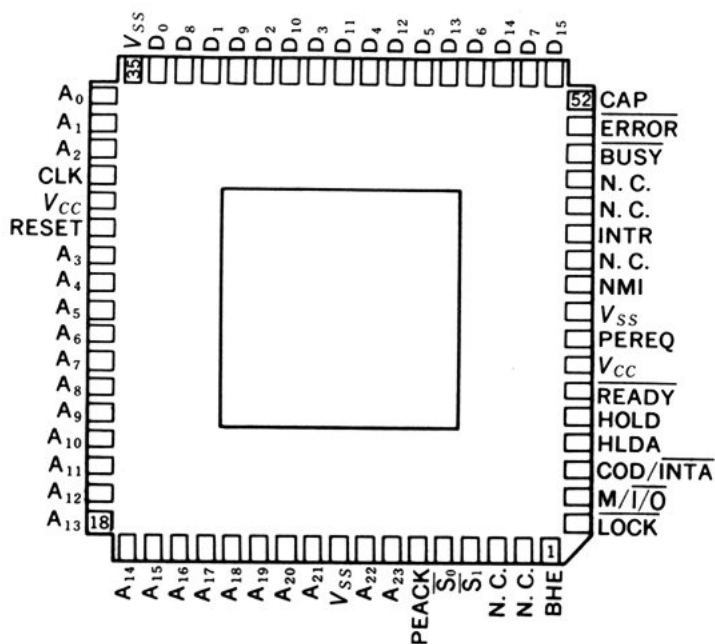



図 1・5 80286 端子配置図

からメモリに送ることによって，メモリがデータを出力するようにハードウェアを設計する．このとき，同時にアドレスバスを使用して，CPU からアドレスをメモリに送る．メモリは CPU から送られたコマンド信号とアドレスに応答して，対応するデータをデータバスへ出力する．この後，CPU はデータバス上の信号を入力し，内部のレジスタに記録する．データライトの場合は，逆に，CPU がデータをデータバスに出力し，メモリがデータバス上の信号を入力する．このようなデータのリード，ライト動作は，I/O の場合もメモリのときと同様に実行される．I/O へのデータのリード，ライトのときには，「I/O からデータをリードする」，または「I/O にデータをライトする」ことを表すコマンド信号が CPU から出力され，アドレスバスで指定された I/O インタフェースが応答する．

80286 は，図 1·4 に示すようにピングリッドアレイパッケージに入っているものと，リードレスチップキャリアパッケージに入っているものとがある．どちらのパッケージも 68 本の端子をもち，このうち 64 本の端子が使用されている．図 1·5 に，リードレスチップキャリアパッケージの場合の端子配置図を示す．

メモリのアドレス信号は $\mathbf{A_0}$-$\mathbf{A_{23}}$ の 24 ビットの端子から出力される．I/O のアドレス信号は，$\mathbf{A_0}$-$\mathbf{A_{15}}$ のアドレスバスの下位 16 ビットから出力される．メモリ，I/O はともに 1 バイト単位で 1 つのアドレスが与えられるから，80286 は **2^{24} バイト（＝16 M バイト）の実メモリ空間**と，**2^{16} バイト（＝64 K バイト）の I/O 空間**をもつ．

データの入出力には $\mathbf{D_0}$-$\mathbf{D_{15}}$ の 16 ビットの端子が使用される．80286 はコマンド信号を直接出力する端子をもたない．そのかわりに，$\overline{S_0}$，$\overline{S_1}$ の端子から出力する 2 ビットのステータス信号によって，80286 からメモリ，I/O に送るコマンドを表現する．80286 がメモリ，I/O に対して何の動作も実行しない状態（アイドル状態）のときは，$\overline{S_0}$，$\overline{S_1}$ はともに High になっている．また，メモリに対する動作か，I/O に対する動作かの区別は M/$\overline{\text{IO}}$ 端子の出力によって区別できる．M/$\overline{\text{IO}}$＝High のときは，メモリに対する動作であることを表し，M/$\overline{\text{IO}}$＝Low のときは，I/O に対する動作であることを表す．$\overline{S_0}$，$\overline{S_1}$，M/$\overline{\text{IO}}$ の信号をデコードすれば，コマンド信号を作ることができる．

1-3 バスサイクル

〔1〕 **バスサイクルの構成**　80286 がメモリ，または I/O からデータのリード，ライトを実行する時間を**バスサイクル**と呼ぶ．80286 が，データリード，データライトを連続して実行する場合のバスサイクルのタイミングを，図 1·6 のタイミングチャートに示す．CLK（クロック）信号は 80286 の CLK 端子から入力される信号で，**システムクロック**と呼ばれる．80286 は，内部でシステムクロックを 2 分周して，**プロセッサクロック**（PCLK）と呼ばれる信号を作る．80286 の内部動作は，プロセッサクロックに同期して実行されるが，80286 の信号のタイミングチャートは，システムクロックを基準にして記述されている．プロセッサクロックが 8 MHz で動作する 80286 には，2 倍の 16 MHz のシステムクロックを供給する．

図 1·6 に示すように，80286 は 2 つのプロセッサクロックで 1 回のバスサイクルを実行する．バスサイクルの中で，最初のプロセッサクロックを**ステータスサイクル**と呼び，記号 T_s で表す．また，次のプロセッサクロックを**コマンドサイクル**と呼び，記号 T_c で表す．T_s では，80286 はステータス信号 $\overline{S_0}$, $\overline{S_1}$ を出力する．

〔2〕 **パイプラインドアドレッシング**　アドレスと M/$\overline{IO}$ は同じタイミングで出力されるが，これらの信号は少なくとも，T_s の 1 システムクロック以前に出力されている．アドレス，M/$\overline{IO}$ は，T_c の ϕ1（フェーズ 1）の最後まで出力され続けることが保証されている．もし，ここで次のバスサイクルが発生したときは，80286 は現在のバスサイクルの実行が終了する，しないに関係なく，T_c の ϕ2 から次のバスサイクルのアドレスと M/$\overline{IO}$ を出力する．

したがって，たとえば図 1·6 のように，リードサイクルのすぐ後に次のライトサイクルが実行される場合は，リードサイクルの T_c の ϕ2 において，リードサイクルの実行が完全に終了していないにもかかわらず，次のアドレスが出力される．

80286 では，データの入出力を実行中に次のバスサイクルのアドレスを出力するように，2 つの連続するバスサイクルの一部を重複させることができ，このことを**パイプラインドアドレッシング**と呼ぶ．パイプラインドアドレッシングの目

的は，メモリの応答スピードがシステムの性能へ影響することをできるだけ小さくすることである．

80286のメモリ空間は16Mバイトの大きさをもち，これは8086のメモリ空間の16倍の大きさとなる．また，バスサイクルの時間も，同じ8MHzのクロックで動作する8086と比較して2倍速くなっている．しかし，プロセッサの性能だけを上げても，メモリの応答スピードが遅ければ，システム全体のスピードの向上は望めない．では，応答スピードの速いメモリを用意すればよいということになるが，コストの問題がある．80286では，パイプラインドアドレッシングによって，比較的低速のメモリを使用した場合も，システム全体の性能をそれほど落とさないように設計することができる．

〔3〕 **コマンドサイクルとウェイトサイクル**　リードサイクルにおいて，80286は T_c の最後に，D_0-D_{15} の端子からデータを入力する．このとき，少なくとも T_c の最後の時間の前後，セットアップタイムとホールドタイムの間，データバス上のデータは安定していなければならない．ライトサイクルにおいては，80286は T_s の $\phi 2$ から，バスサイクルの1システムクロック後までの間，データを D_0-D_{15} の端子に出力する．

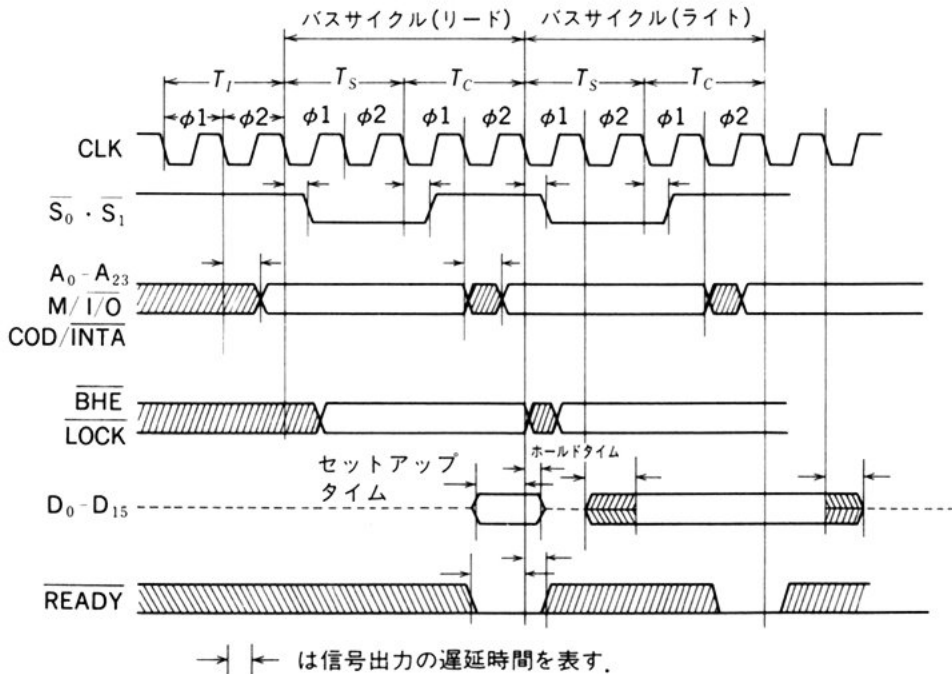



図 1·6　基本バスサイクル

リードサイクルの場合，メモリは T_c の最後からセットアップタイム前までにデータを出力するように応答しなければならない．また，ライトサイクルの場合，メモリは T_c の最後から1システムサイクル後までにデータを入力するように応答しなければならない．もし，正しくデータのリード，ライトができないような応答スピードの遅いメモリを使用する場合は，T_c の後に任意の数の T_c を挿入して，バスサイクルを延長することができる．

このとき，データのリードはバスサイクル最後の T_c の終わりで実行される．また，ライトサイクルにおいて，最後の T_c の1システムクロック後までデータを出力する．バスサイクルを延長するために挿入する T_c を**ウェイトサイクル**と呼ぶ．バスサイクルにウェイトサイクルを入れるか，入れないかは $\overline{\text{READY}}$（レディ）信号によって制御できる．バスサイクルの最後に $\overline{\text{READY}}$＝Low であれば，バスサイクルは終了するが，逆に，$\overline{\text{READY}}$＝High のときは，T_c の後に，再び T_c が繰り返される．$\overline{\text{READY}}$ を High のままにしておけば，ウェイトサイクルは繰り返され，バスサイクルは終了しない．

1-4 レジスタ構成

次に，80286の内部構成について考える．アセンブリ言語でプログラムする上で知っておくべきことは，80286がどのようなレジスタ構成をもつかということである．80286のレジスタ構成を図1·7に示す．また，**FLAG, MSW**のそれぞれのビット構成を図1·8(a)，(b)に示す．**汎用レジスタ**と呼ばれる一連のレジスタ群の種類および扱い方は，まったく8086の汎用レジスタと同じである．

ワード(16ビット)汎用レジスタには，AX，BX，CX，DX，BP，SP，SI，DIの8本のレジスタがある．8086，80286において，このように，汎用レジスタ名が統一されていないのは，ある種の命令では，自動的にレジスタの役目が決まっているからである．各レジスタは，その特別な役目を表す名前をもっている．たとえば，CXはカウンタレジスタを表す．これは，LOOP命令，ストリング命令において繰り返し処理の回数を指定するために使用される．

特定の命令において使用するレジスタを暗黙的に定義しておけば，命令コードを短くすることができる．逆に，レジスタの使用に自由がなくなるが，プログラムを書くとき，結局はプログラマがレジスタの使用目的を決めるのだから，レジスタ使用の自由度が小さいことはそれほどの欠点にはならない．

セグメントレジスタの種類も8086のものと同じである．しかし，80286のセグメントレジスタには，それぞれ，48ビットの**セグメントキャッシュ**が付属している．セグメントキャッシュはセグメントの属性を記録し，80286がメモリ管理機能を実行するときに内部的に使用するレジスタであるが，セグメントキャッシュには直接に値を代入することはできない．

FLAGは図1·8(a)に示したようなビットに区分され，それぞれのビットが演算結果の状態を表したり，また，80286の動作を制御するために使用される．FLAGのビット構成はビット0からビット11までが，8086のFLAGと同じであるが，新たに**IOPL**と**NT**が追加されている．

MSW(マシンステータスワード)はFLAGと似た性格のレジスタであり，図1·8(b)に示すように，下位4ビットが使用されている．MSWはFLAGよりももっと，80286の動作に根本的に影響するフラグを表す．

ワード(16ビット)汎用レジスタ

15 0

AX AH AL

BX BH BL

CX CH CL

DX DH DL

バイト(8ビット)汎用レジスタ

BP

SP

SI

DI

15 0

FLAG

MSW

セグメントレジスタ

15 0 47 0

CS CS キャッシュ

DS DS キャッシュ

SS SS キャッシュ

ES ES キャッシュ

39 0

GDTR

IDTR

15 0

LDTR LDTRキャッシュ

TR TRキャッシュ

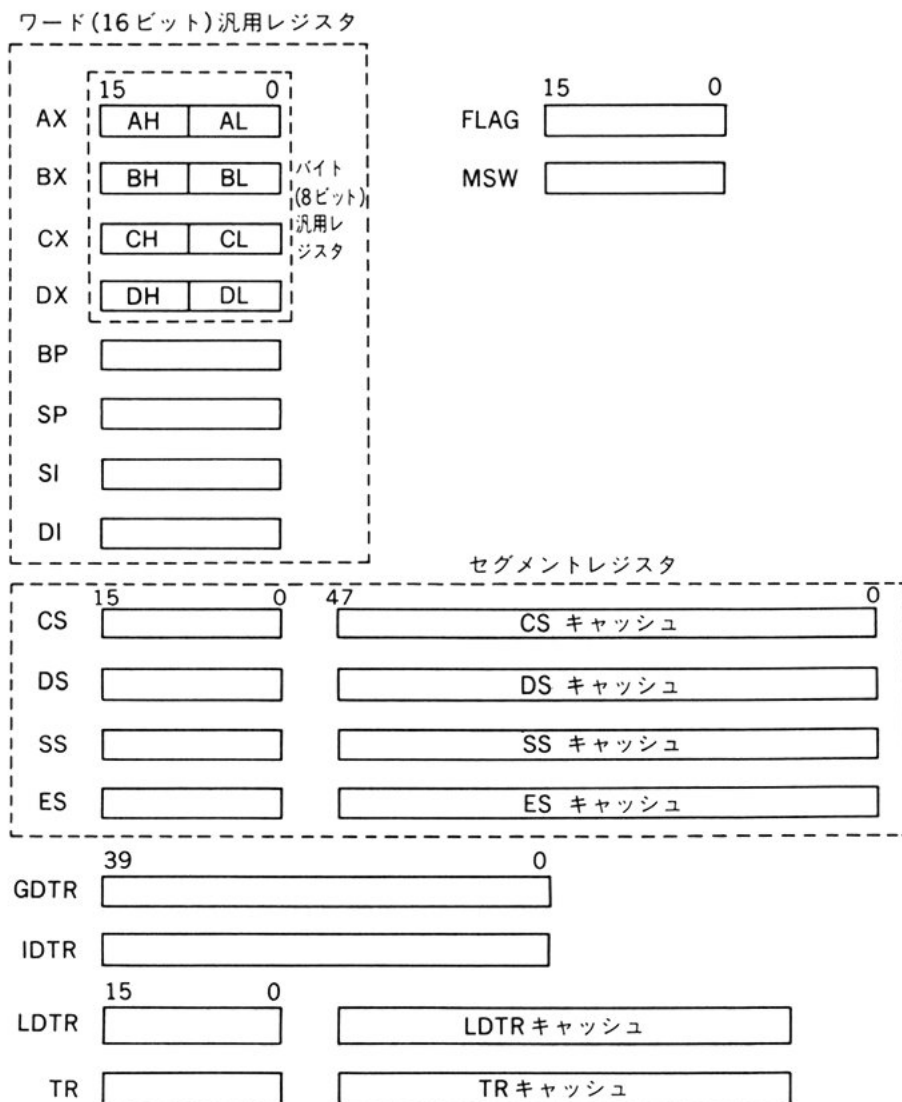



図 1・7 80286のレジスタ構成

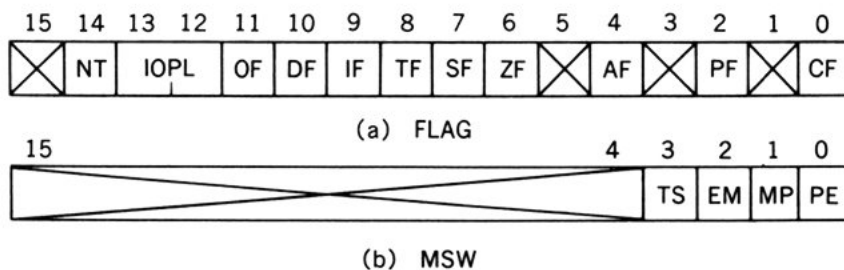



図 1・8 FLAG, MSWのビットフィールド

1-5 セグメンテーション

〔1〕 **セグメントと論理アドレス**　80286のアドレスバスは24ビットであるから，$2^{24}=16$ Mバイトのメモリ空間をもつ．(ただし，後に述べるように，リアルアドレスモードで使用するときは，アドレスバスの下位20ビットだけを使用するので，$2^{20}=1$ Mバイトのメモリ空間をもつ)．80286はメモリのリード，ライトにおいて端子A_0-A_{23}に24ビットのアドレスを出力する．アドレスバスに出力される24ビットの信号を**物理アドレス**と呼ぶ．

しかし，80286内部においては(すなわち，プログラムにおいては)物理アドレスを直接使用することはない．

プログラムは，命令を定義したコード領域，変数を定義したデータ領域，そしてスタック領域から構成され，図1·9に示すようにメモリに配置される．これらの領域は，それぞれ，メモリの連続した領域に定義されるが，領域相互間は必ずしも連続したメモリに配置する必要はない．

このように領域を相互に独立に扱うためには，メモリ空間内の領域の位置を表すアドレスと，領域内のデータの位置を表すアドレスの，2つのアドレスによってメモリ空間内の特定データの位置を表現すればよい．

ここで，独立に扱うことのできる領域を**セグメント**と呼ぶ．メモリ空間内のセグメントの位置を表すアドレスを**セグメントセレクタ**と呼ぶ．また，セグメント内の特定のデータの位置を表すアドレスを**オフセット**(**相対アドレス**)と呼ぶ．そして，セグメントを使用した2次元的なアドレス指定法を**セグメンテーション**と呼ぶ．セグメントセレクタとオフセットの組み合わせによってメモリ空間内の任意のバイト，ワードデータを一意に指定することができる．セグメントセレクタとオフセットを**論理アドレス**と呼ぶ．

〔2〕 **セグメントレジスタの使用**　オフセットは，メモリを参照する命令の中で指定するが，セグメントセレクタは命令の中で直接，指定するのではなく，あらかじめセグメントレジスタに初期設定しておく．セグメントレジスタは図1·10に示すように，CS，SS，DS，ESの4種類があり，メモリ参照の種類によって使用されるセグメントレジスタが，表1·2のように決められている．80286は命令をリードする(コードフェッチ)ときは，必ずCSによって指定されるセ

グメントから命令をリードする．このとき，オフセットは必ずIPの値を使用する．PUSH，POP操作によって，データを保存したり，または，保存したデータを読み出したりするときは，セグメントセレクタに必ずSSが使用される．

また，オフセットはSPによって指定される．オフセットを表現するために，

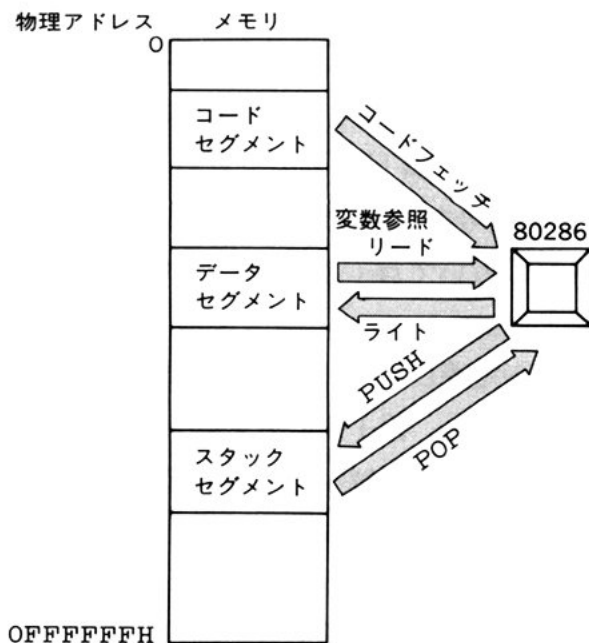



図 1・9　80286 メモリ空間

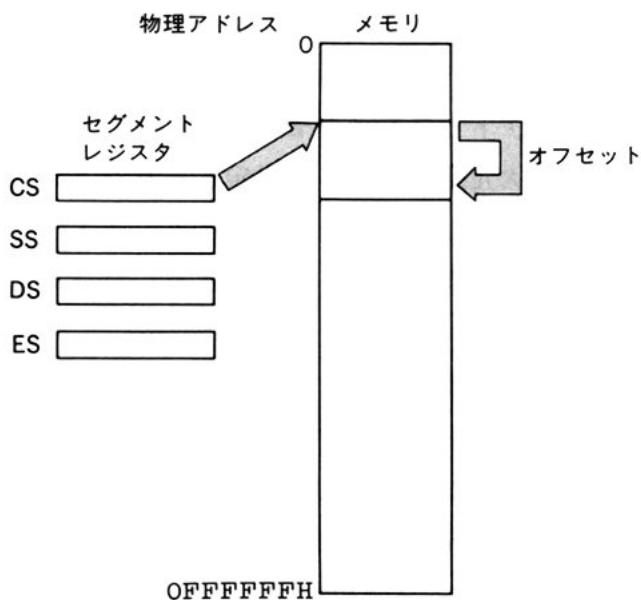



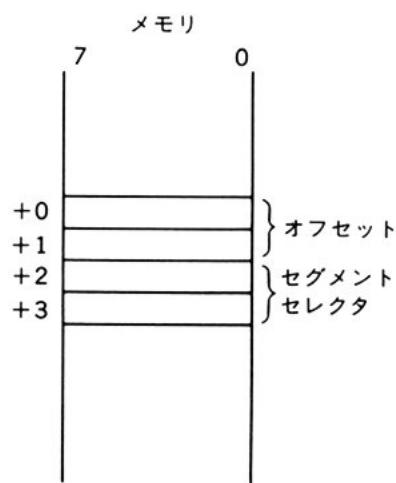
図 1・10　セグメントレジスタとメモリの関係

表 1・2 セグメントレジスタの使用

メモリ参照の形式	使用セグメントレジスタ	オフセット
コードフェッチ	CS	IP
スタックへの PUSH，POP	SS	SP
BP を使用した間接的データ参照	SS	命令のオペランドで指定する． ただし，BP を使用して，間接的にオフセットを指定する場合に限る． また，セグメントオーバーライドプリフィックスを命令の前に付けることによって，SS 以外の DS，ES，CS の任意のセグメントレジスタを使用することが可能である．
データ参照	DS	命令のオペランドで指定する． ただし，BP を使用した間接的オフセット指定は除く． また，セグメントオーバーライドプリフィックスを命令の前に付けることによって，DS 以外の ES，SS，CS の任意のセグメントレジスタを使用することが可能である．

BP を間接的に使用してデータ参照を行う場合も，セグメントセレクタを指定するために SS が使用される．BP を間接的に使っていないデータ参照は DS がセグメントセレクタを指定するために使用される．しかし，データ参照の場合は，このようにセグメントレジスタの使い方が固定されたままでは，使いにくい．したがって，命令の前に，**セグメントオーバライドインストラクションプリフィックス**と呼ばれる 1 バイトのコードを追加することによって，暗黙的なセグメントレジスタの使用規則にかかわらず，DS，ES，SS，CS の中から任意のセグメントレジスタを使用することができる．

図 1・11 ポインタ

重要なことは，命令の中でセグメントセレクタの値を直接指定するのではなく，セグメントレジスタの種類を指定することである．このことは大きな利益をもたらす．セグメントレジスタに，最初，セグメントセレクタの値を代入してお

けば，後はオフセットだけを指定することによって，一度にメモリを参照することができる．たとえば

```
MOV AX , WORD PTR 10
```

はDSで指定されるセグメントのオフセット10から始まる1ワードのデータをAXに転送する命令である．AXに転送する値がメモリ内のセグメントセレクタ＝123，オフセット＝10（123：10のように表現する）に定義されている場合は，この命令を含むプログラムを実行する前に，DSに123を初期設定しておく必要がある．

次に，このデータセグメントがセグメントセレクタ＝234で指定されるセグメントに再配置された場合を考えると，上の命令を含めて，プログラムを書き換える必要はない．DSに123ではなく，234を初期設定するだけでよい．

プログラムにおいて，セグメントセレクタとオフセットの両方の値を指定しなければならない場合もある．このとき，図1･11のように，メモリにオフセットとセグメントセレクタとを連続に定義して使用できる．この32ビットのデータをポインタと呼ぶ．

1-6 リアルモードとプロテクトモード

〔1〕 **80286 動作モード**　80286 は図 1·12 に示すように 2 つの動作モードをもつ．一方を**リアルアドレスモード**と呼び，他方を**プロテクティドバーチャルアドレスモード**と呼ぶ．リアルアドレスモードという言葉は呼びにくいので，**リアルモード**と呼ぶことにする．リアルモードでは，80286 を高速の 8086 として使用することができる．このため，リアルモードはまた，**86**（ハチロク）**モード**と呼ばれることもある．ただし，リアルモードでは，80286 のすべての機能を使用することができない．たとえば，アドレスバスは 24 ビットのうち，下位 20 ビットしか使用できない．したがって，リアルモードでのメモリ空間は 8086 と同様の 1M バイトの大きさとなる．

80286 の強力な能力を使用するためにはプロテクティドバーチャルアドレスモードで使用する．このモードの名前も長く呼びにくいので，**プロテクトモード**と呼ぶことにする．プロテクトモードは，80286 固有の特徴を引き出せるモードであるため，**286**（ニーハチロク）**モード**と呼ぶこともある．リアルモード，プロテクトモードの制御は，MSW の PE によって実行する．PE＝0 のとき，80286 はリアルモードで動作し，PE＝1 のとき，プロテクトモードで動作する．

図 1·12 に示したように，80286 をリセットした直後は PE＝0 であり，リアルモードで動作する．リアルモードからプロテクトモードに変更するためには MSW の PE を 1 にする．MSW は図 1·13 に示す LMSW 命令を使用して，データをロードする（MSW へ書き込む）こともできるし，逆に，SMSW 命令を使用して，MSW のデータをストアする（MSW から読み出す）こともできる．MSW の PE だけを 1 に変更するためには，図 1·14 に示すような処理を実行する．ただし，ここに示した処理を実行するだけではプロテクトモードでの正しい実行はできない．プロテクトモードへの完全な初期化処理は **3-7** で述べる．

〔2〕 **リアルモードの論理アドレス**　リアルモードとプロテクトモードの違いは表 1·3 のようにまとめることができる．命令体系，機能においては，プロテクトモードはリアルモードの命令体系，機能に新しいものが追加された形になっている．表 1·3 に示すそれぞれの特徴の中で，最も顕著なものはメモリ空間の大きさである．

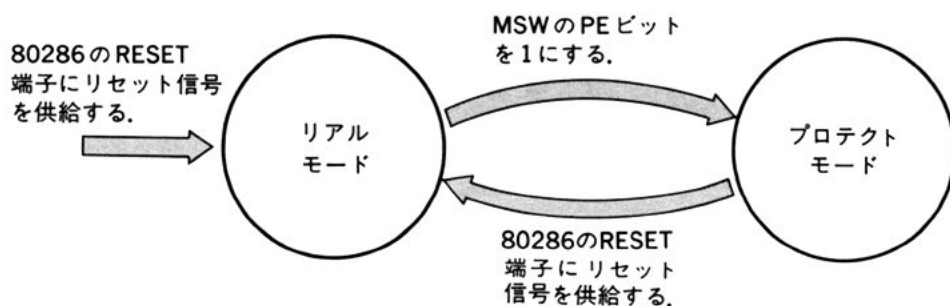



図 1・12 リアルモードとプロテクトモード

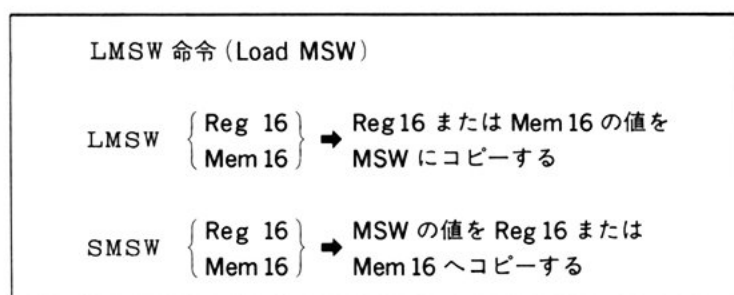



図 1・13 LMSW 命令と SMSW 命令

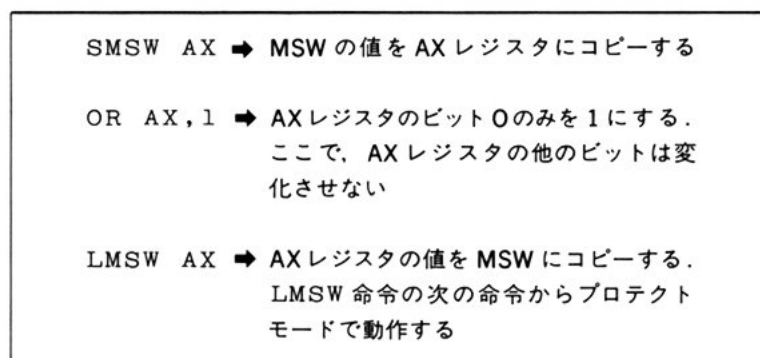



図 1・14 リアルモードからプロテクトモードへの切り換え

80286 内部ではメモリのアドレスを表現するとき，論理アドレスを使用する．80286 は論理アドレスから 24 ビットの物理アドレスを作り，アドレスバスへ出力するのであるが，この方法はリアルモードの場合と，プロテクトモードの場合とではまったく異なる．ここでは，リアルモードにおける，論理アドレスから物理アドレスへの変換について述べる．

セグメントレジスタには，それぞれ，図 1·15 に示すようなセグメントキャッシュが付属する．セグメントキャッシュは 48 ビットのレジスタで，16 ビットのリミットフィールド，24 ビットのベースアドレスフィールド，そして，8 ビット

表 1・3　リアルモードとプロテクトモードの基本的特徴

項　　目	リアルモードの特徴	プロテクトモードの特徴
データバス	16 ビット	16 ビット
アドレスバス	20 ビット（A_{19}-A_0 を使用） メモリ空間＝1M バイト	24 ビット（A_{23}-A_0 を使用） 実メモリ空間＝16M バイト
使用レジスタ	AX，BX，CX，DX SI，DI SP，BP，IP FLAG CS，DS，SS，ES MSW，GDTR，IDTR	AX，BX，CX，DX SI，DI SP，BP，IP FLAG CS，DS，SS，ES MSW，GDTR，IDTR LDTR，TR
命令と機能	80186 と共通の命令 ＋ LGDT 命令，SGDT 命令 LIDT 命令，SIDT 命令 LMSW 命令，SMSW 命令 CLTS 命令	80286 リアルモードと共通の命令 ＋ LLDT 命令，SLDT 命令 LTR 命令，STR 命令 LAR 命令，LSL 命令 VERR 命令，VERW 命令 ARPL 命令 特権レベルによるメモリ，I/O 管理 タスクスイッチ命令 仮想メモリのためのサポート

	セグメントレジスタ	7　　0	23　　0	15　　0
CS	セグメントセレクタ		セグメントセレクタ×16	0FFFFH
SS	セグメントセレクタ		セグメントセレクタ×16	0FFFFH
DS	セグメントセレクタ		セグメントセレクタ×16	0FFFFH
ES	セグメントセレクタ		セグメントセレクタ×16	0FFFFH

AR フィールド　ベースアドレスフィールド　リミットフィールド

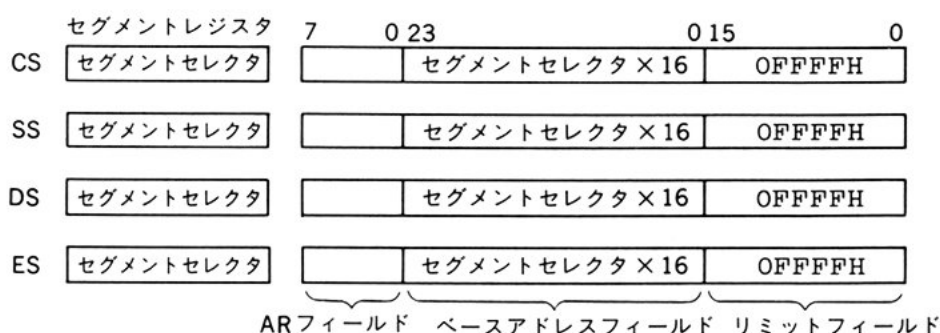



図 1・15　リアルモードにおけるセグメントレジスタとセグメントキャッシュ

の AR（アクセスライト）フィールドに分離される．

リミットフィールドはセグメントの最大のオフセットを定義するが，リアルモードでは，0FFFFH に固定であり，変更することはできない．ベースアドレスフィールドは，セグメントのオフセット 0 の物理アドレスを定義する．80286 はベースアドレスフィールドの値にオフセットを加算した物理アドレスをアドレス

バスに出力する．しかし，リアルモードではベースアドレスフィールドの値を任意に定めることはできず，必ず，セグメントレジスタの値を16倍した値が自動的に設定される．セグメントレジスタが書き換えられたときに，自動的にベースアドレスフィールドも更新される．したがって，リアルモードにおいて，8086のように論理アドレス OFFFOH：0400H が論理アドレス 0000H：0300H と同じ物理アドレスを指定するようにするためには，A_{20}-A_{23} の4ビットが常に0になるように，外部にハードウェアを付加しなければならない．

AR フィールドにはセグメントの属性を表すコードが入るが，リアルモードにおいては使用されない．

8086の論理アドレス

8086および80286は本質的にリロケータブル（再配置可能）なプログラムができるように作られている．リロケータブルとは，プログラムをメモリの別の位置に移し換えても，書き換えることなく動作することを意味する．

その手段としてセグメンテーションが8086から採用された．プログラムでメモリを参照するとき，アドレスをある基準位置からのオフセット（相対アドレス）で表す．メモリの基準位置（ベース）はセグメントレジスタと呼ばれる4種類の16ビットレジスタに定義し，ベースからオフセットで参照できる領域をセグメントと呼ぶ．したがって，8086の任意のアドレスはセグメントとオフセットの組み合わせによって表現され，これを論理アドレスと呼ぶ．

論理アドレスから物理アドレスの計算は，セグメントの値を4ビットだけ左シフトしてからオフセットを加算する．このため，8086のセグメントのベースアドレスの下位4ビットは必ず0である．したがって物理アドレスの0番地から16バイトごとにセグメントのベースを決めることができる．8086ではこの16バイトの境界をパラグラフと呼び，セグメントレジスタの値をパラグラフ番号とも呼ぶ．

2. リアルモードでの使用

リアルモードで使用する80286は，第1章で述べたように高速の8086として使用することができる．8086のプログラミングに習熟したプログラマは，80286をリアルモードで使用する場合にも同様にプログラムすることができる．8086で動いていたプログラムを，そのままで80286上で走らせることもできる．ここでは，リアルモードで使用できる80286の命令について説明する．この章で述べる命令はプロテクトモードでも使用でき，リアルモード，プロテクトモードの両モードにおいてプログラムを書くための基本的な命令である．

2-1　メモリ，I/O参照と転送命令

〔1〕 **メモリ，I/Oのアドレス指定**　80286で扱うことのできるデータには，図2·1に示すように8ビットのバイト，16ビットのワード，32ビットのダブルワードがある．ダブルワードは主にポインタデータを格納するために使用される．2バイト以上のデータをメモリに配置するときは，すべて下位バイトを小さいアドレスに，上位バイトを大きいアドレスに連続に配置する．I/O空間にはインタフェースのレジスタが並んでいるものと考えればよい．I/Oとの間では，図2·2に示すようにバイト，ワードデータを入出力することができる．ワードデータは，やはり下位バイトがI/Oの小さいアドレスに，上位バイトがI/Oの大きいアドレスになるよう配置する．

メモリのオフセットを指定する方法は，図2·3に示すような方法がある．図2·3(1)に示す方法は**直接指定**と呼び，直接に定数でオフセットを指定する．アセンブリ言語の命令では，データのタイプも表現する必要があるので

BYTE　PTR　10　➡ オフセットが10のメモリに配置したバイトデータ
WORD　PTR　10　➡ オフセットが10のメモリに配置したワードデータ
DWORD　PTR　10　➡ オフセットが10のメモリに配置したダブルワードデータ

のように書く．ここで，大きさが2バイト以上のデータのオフセットは，そのもっとも下位のバイトのオフセットで表す．図2·3(2)に示す**間接指定**は，**ベースレジスタ**または**インデックスレジスタ**を用いて，間接的にオフセットを指定する方法である．ベースレジスタとは，BX，BPの総称であり配列などの構造的データの基底アドレスを表すのに用いられる．インデックスレジスタは，SI，DIの総称であり，配列，ストラクチャなどの構造的データの各要素を間接的に表すために用いられる．間接指定はベースレジスタ，インデックスレジスタ，ディスプレイスメント（定数で表した変位）の3要素の任意の組み合わせによって，さまざまな構造のデータを柔軟に指定することができる．間接指定の3要素のうち，指定された要素の和がオフセットになる．

I/O にはセグメンテーションはなく，0 から 65535 までの I/O アドレスを指定する．I/Oアドレスの表現も，図 2･4 に示すように直接指定と間接指定があるが，直接に指定できるのは，0 から 255 までである．

〔2〕 **転送命令**　　レジスタ，メモリ，I/O のデータ転送は，図 2･5 に示すような命令で行われる．メモリ-レジスタ間，レジスタ-レジスタ間のデータ転送は MOV 命令または XCHG 命令を使用する．MOV 命令，XCHG 命令の書き方を図 2･6 に示す．MOV 命令は 2 つのオペランドをもち，オペランド 2 の値をオペ

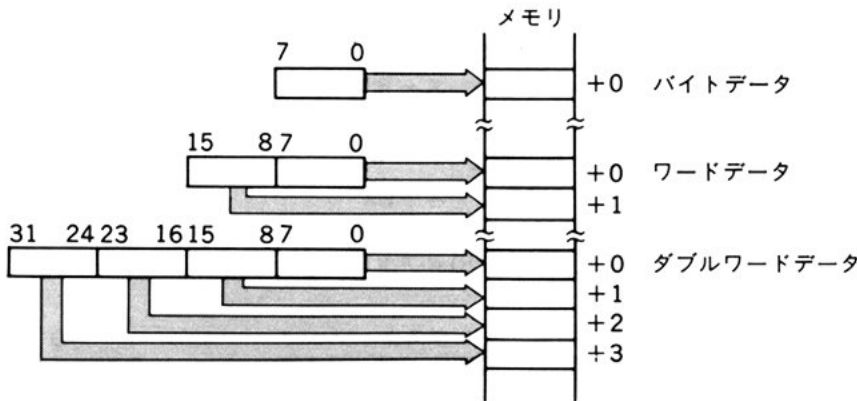



図 2･1 80286 のデータとメモリへの配置

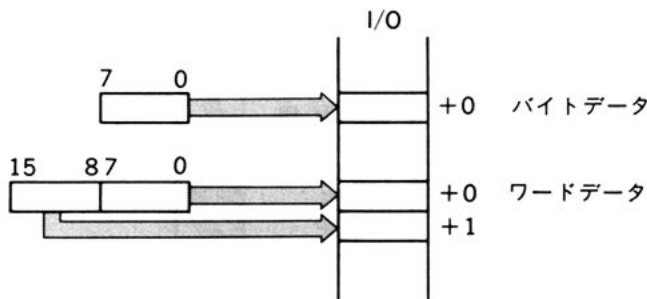



図 2･2 I/O へのデータの配置

（1） **直接指定**
オフセット＝定数

（2） **間接指定**
オフセット＝[ベースレジスタ]＋[インデックスレジスタ]± ディスプレイスメント
ベースレジスタ＝BX または BP
インデックスレジスタ＝SI または DI
ディスプレイスメント＝ベースレジスタ＋インデックスレジスタからの相対変位を表す符号付き定数

図 2･3 アドレッシングモード

ランド1に代入する．オペランド1にDS, ES, SSのセグメントレジスタを指定することもできるが，このときオペランド2に定数を指定することはできない．左側のオペランドが値を代入する先（デスティネーション）を表す．インテルのアセンブリ言語の命令では，MOV命令に限らず2つのオペランドをもち，どちらかが値を更新される場合，値を更新するオペランドを必ず左側に書くようにする．ここで，オペランドは，バイトタイプでもワードタイプでもよいが，両方のオペランドのタイプは一致していなければならない．また，両方のオペランドに

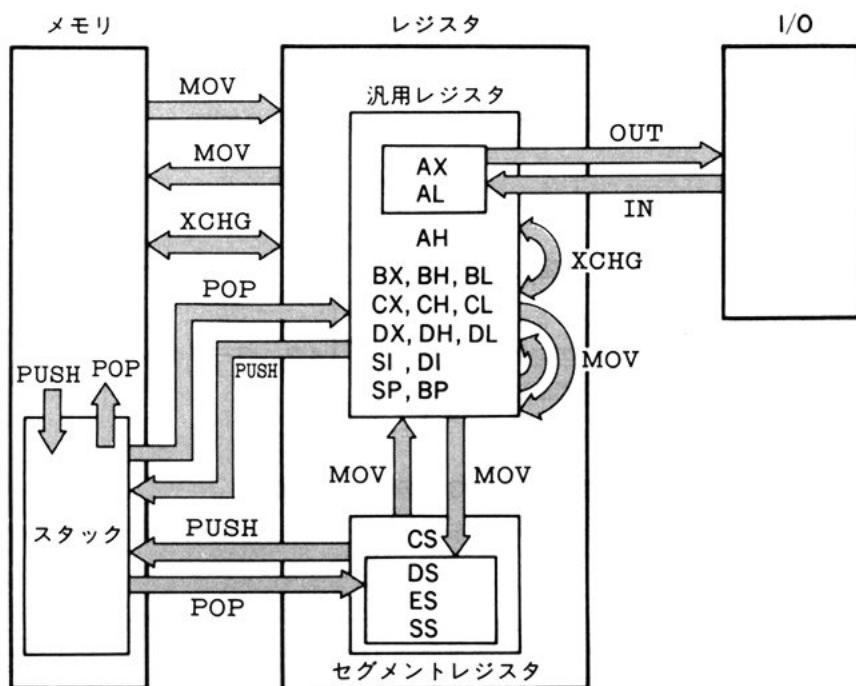
図 2・4　I/O のアドレス指定

ただし，転送するデータのタイプはバイトまたはワード（＝2バイト）であり，転送元と転送先のデータタイプは一致しなければならない．
また，スタックへのPUSH, POPで転送されるデータタイプは常にワードである．

図 2・5　データ転送

同時にメモリを指定することは許されない．XCHG 命令は，2 つのオペランドの値を相互に入れ換える命令である．

〔3〕 **スタックの使用**　図 2･5 において，スタックはデータを保存するためのセグメントであるが，図 2･7 に示すように，PUSH，POP と呼ばれる方法で

（1） **MOV　オペランド 1，オペランド 2**
オペランド 1 = バイトメモリ，ワードメモリ，
バイト汎用レジスタ，ワード汎用レジスタ
DS，SS，ES
オペランド 2 = バイトメモリ，ワードメモリ，
バイト汎用レジスタ，ワード汎用レジスタ
DS，SS，ES，CS，定数

（2） **XCHG　オペランド 1，オペランド 2**
オペランド 1 = バイトメモリ，ワードメモリ，
バイト汎用レジスタ，ワード汎用レジスタ
オペランド 2 = バイトメモリ，ワードメモリ，
バイト汎用レジスタ，ワード汎用レジスタ

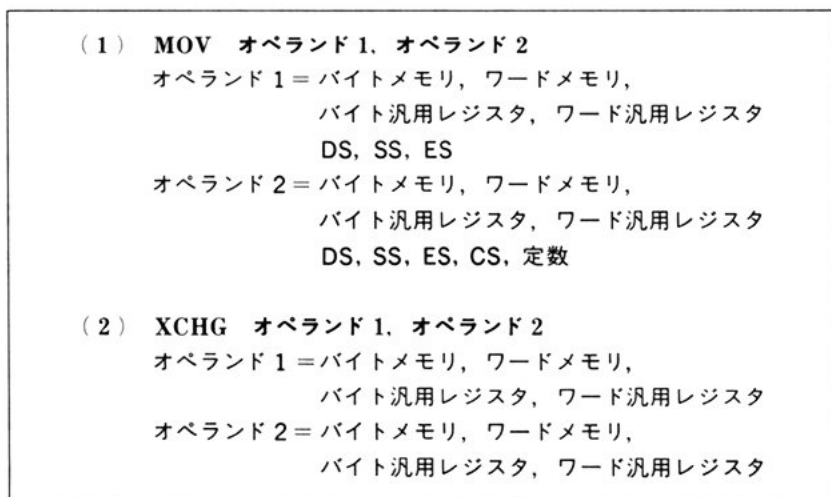
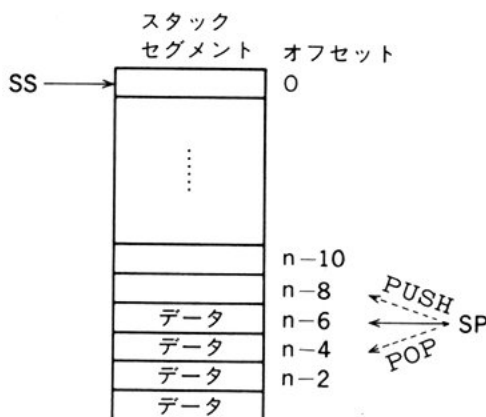
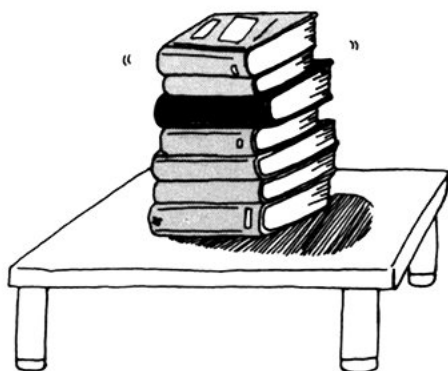



図 2･6　MOV，XCHG 命令

（1） **PUSH　オペランド**
オペランド = ワードメモ，リワード汎用レジスタ
CS, DS, ES, SS, 定数
PUSH の操作：SP←SP − 2
WORD PTR SS：[SP]← **オペランド**

（2） **POP　オペランド**
オペランド = ワードメモリワード汎用レジスタ
DS, ES, SS
POP の操作：オペランド← WORD PTR SS：[SP]
SP←SP + 2

図 2･7　スタックと PUSH，POP 操作

データを出し入れする．スタックは，机の上に本を積むようにデータを保存する．このようにすると，スタックのトップに最も新しいデータを見ることができる．スタックセグメントはオフセットの高い領域から低い領域へと使用する．スタックトップのオフセットはSPによって指定する．PUSHはスタックに新しいデータを積む操作である．このとき，SPは自動的に2だけ減算される．POPは逆にスタックトップのデータを取り出す操作である．この場合，SPは2だけ加算される．PUSH，POPの操作は，PUSH命令，POP命令を使用してデータを保存するときに実行される．また，CALL命令，割り込み，RET命令，IRET命令で，主プログラムへの戻りアドレスを保存するために自動的にPUSH，POPが実行される．PUSH，POPで扱われるデータは必ずワードタイプでなければならない．

〔4〕 **I/Oへのデータ転送**　I/Oとのデータの入出力は，図2・8に示すようにIN命令，OUT命令を使用する．I/Oアドレスは直接に定数で指定してもよいし，DXで間接的に指定することもできる．ただし，定数でI/Oアドレスを指定するときは0から255までのアドレスしか指定できない．IN命令，OUT命令では，バイト，ワードの2種類のデータを扱うことができるが，このタイプの区別は，命令のオペランドにALを使用するかAXを使用するかによって区別できる．

本のスタック

（1） **IN命令**

IN $\left\{\begin{matrix} \text{AL} \\ \text{AX} \end{matrix}\right\}$, I/Oアドレス

（2） **OUT命令**

OUT I/Oアドレス, $\left\{\begin{matrix} \text{AL} \\ \text{AX} \end{matrix}\right\}$

図2・8　IN命令とOUT命令

2-2 演 算 命 令

演算命令には算術演算，論理演算，シフト，ローテイトがある．算術演算の中で，加算，減算は図 2･9 に示すものがある．ADC 命令，SBB 命令はキャリーフラグ（CF）を含めた演算で，32 ビットのデータの加算，減算などに使用する．たとえば，オフセットが 0 のメモリにある 32 ビットデータと，オフセットが 4 のメモリにある 32 ビットデータを加算し，結果をオフセットが 8 のメモリに代入する処理は，図 2･10 に示すようになる．32 ビットデータの演算では，下位ワード，上位ワードを別々に処理するが，上位ワードの演算において下位ワードからの桁上げも含めて演算しなければならない．したがって，上位ワードの加算は，下位ワードの加算の後に実行し ADC 命令を使用する．このことは，減算の場合も同様である．減算のときは，下位ワードの減算で引きすぎが発生した場合，CF が 1 となるから，上位ワードの減算には SBB を使用すればよい．

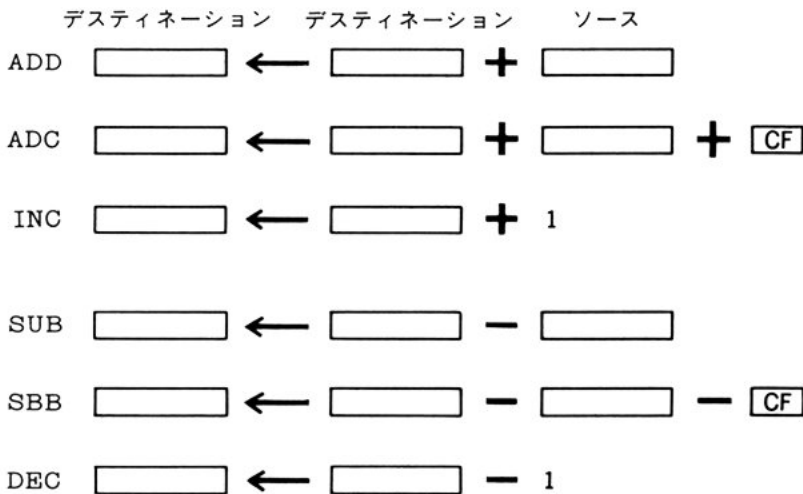



図 2・9　加算と減算

2　リアルモードでの使用

乗算と除算の命令を図2･11に示す．MUL命令，DIV命令では，データは符号なしとして扱われる．IMUL命令，IDIV命令では，データの最上位ビットが符号ビットで，マイナスデータは2の補数表現で扱われる．また，IMULにはワード汎用レジスタ，ワードメモリに置かれた符号付き整数を定数で乗算できるものがあり，この命令は3つのオペランドをもつ．ただし，左側の2つのオペランドが同じレジスタの場合は，右側の2つのオペランドだけを指定する．

ローテイト，シフト演算には，図2･12に示す種類がある．命令の書き方はす

```
MOV AX , WORD PTR 0
MOV DX , WORD PTR 2
ADD AX , WORD PTR 4
ADC DX , WORD PTR 6
MOV WORD PTR 8 , AX
MOV WORD PTR 10, DX
```

図 2･10　32ビットデータの加算

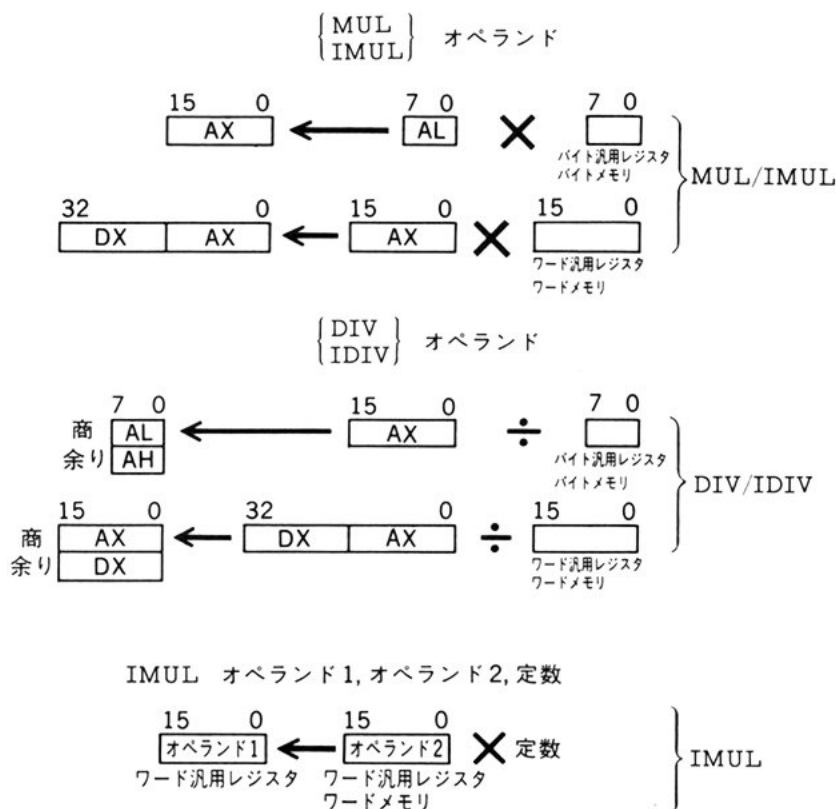



図 2･11　乗算と除算

べて同じである．図2·12に示すように，オペランド1にはバイト，またはワードタイプの汎用レジスタ，あるいはメモリを指定し，右のオペランドにローテイト，またはシフトするビット数を指定する．ビット数は定数で直接指定してもよいし，CLで間接的に指定することもできる．

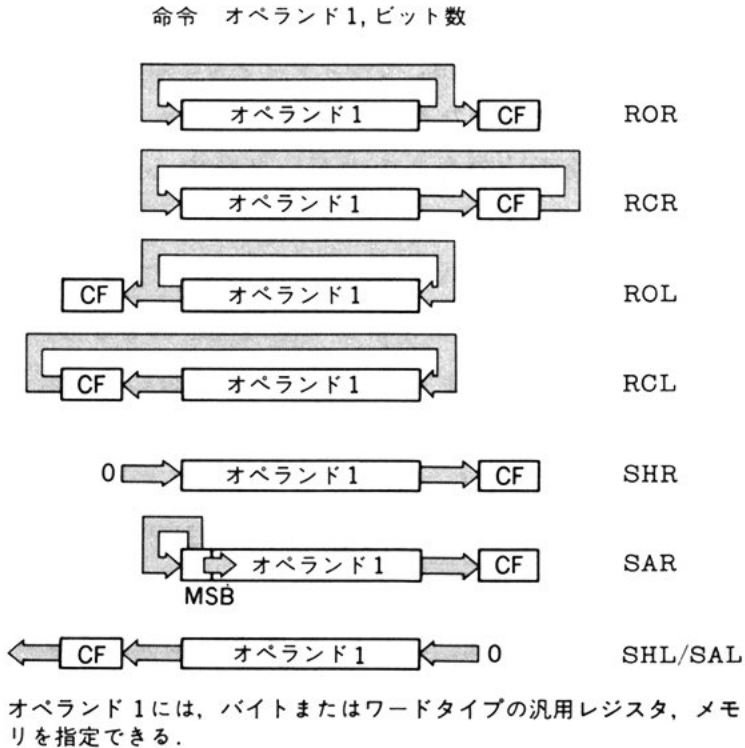



図 2·12　ローテイト，シフト演算

2-3 制御命令

プログラムの処理の流れを変える命令には，図2·13に示すようにJMP命令，CALL命令，RET命令がある．これらの命令はその機能によって，さらに何種類かに分類される．

〔1〕 **JMP命令**　　JMP命令は，そのジャンプできる距離によって，3種類に分類できる．それを，識別するために，**short**，**near**，**far**の言葉を使用する．short JMPは，JMP命令の次の命令のアドレスを中心に＋127～－128バイトの範囲にジャンプすることができる．near JMPは，JMP命令の次の命令のアドレスを中心に$(2^{15}-1)\sim -2^{15}$バイトの範囲にジャンプすることができる．オフセットの計算ではビット15からのキャリーは無視されるから，near JMPを使用すればセグメント内の任意のアドレスへジャンプすることができる．far JMPは，命令の中にジャンプする先のセグメントセレクタとオフセットを指定し，80286のメモリ空間の任意のアドレスにジャンプすることができる．short JMP，near JMPがIPしか書き換えないのに対して，far JMPはCSとIPを書き換えてジャンプする．

〔2〕 **CALL命令とRET命令**　　CALL命令にもJMP命令と同様にnearとfarの区別がある．CALL命令は，指定されたアドレスにジャンプする前にCALL命令の次の命令のアドレスを戻りアドレスとしてスタックへPUSHする．near CALLでは，near JMPと同様にCSの値は変化しないから，図2·14(a)に示すように戻りアドレスとしてオフセットだけをPUSHする．far CALLの場合はCSとIPの値がともに変化するから，図2·14(b)に示すようにセグメントセレクタ，オフセットの順で2ワードの戻りアドレスをPUSHする．

CALL命令で制御を移した手続きの中でRET命令を実行すると，CALL命令の次の命令へ戻ることができる．CALL命令にnearとfarの2種類があるのに対して，RET命令にもnear, farの2種類がある．near RETはスタックトップから1ワードをPOPして，IPに代入することによって制御を変える．far RETは，スタックトップから2ワードをPOPしてIPとCSに代入して制御を変える．したがって，near CALLに対してnear RET，far CALLに対してfar RETのように対にして使用しなければ正しい制御の変更はできない．ここでは，JMP

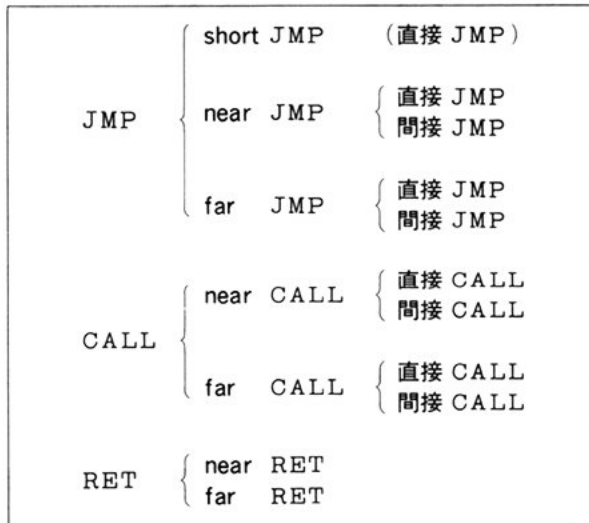



図 2・13　JMP 命令，CALL 命令，RET 命令の種類

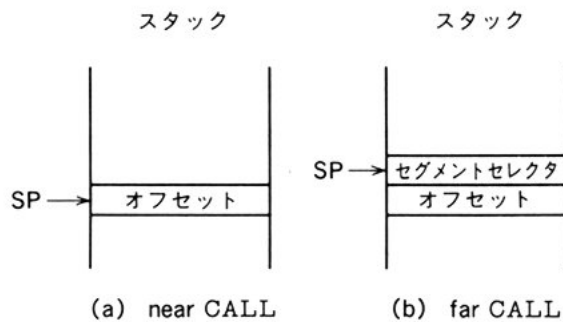



図 2・14　CALL 命令実行後のスタック

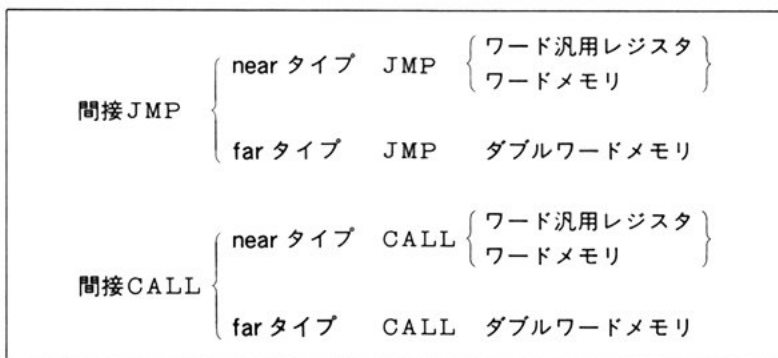



図 2・15　間接 JMP 命令と間接 CALL 命令

命令，CALL 命令，RET 命令のタイプを明確に表現するために，命令の前に short，near，far と記したが，アセンブリ言語 ASM 286 では short，near，far を書く必要はない．

〔3〕 **間接的なJMP, CALL命令**　JMP命令，CALL命令にはそれぞれ，間接JMP，間接CALLと呼ばれるものがある．この命令の書き方を図2·15に示す．これらの命令は，オペランドに指定したワード汎用レジスタまたは変数によって，間接的に制御を移すアドレスを指定する．間接JMP，間接CALLにもnearタイプのものとfarタイプのものがある．オペランドにワード汎用レジスタまたはワードメモリを指定した命令はnearタイプであり，そのレジスタまたはメモリの値をIPに代入して制御を移す．オペランドにダブルワードメモリを指定したものはfarタイプであり，ダブルワードメモリの上位ワードをCSに代入し，下位ワードをIPに代入して制御を移す．

〔4〕 **条件JMP命令**　アセンブリ言語のプログラムで分岐処理を定義する命令が条件JMPである．表2·1に単一フラグによる条件JMP命令を示す．状態フラグの値によってオペランドに指定したラベルへジャンプする．フラグの値が条件と逆の場合は次の命令を実行する．プログラムでは2つの値の大小比較の結果によって，分岐する処理を書くことが多い．そのような処理を簡単に表現できるように2つ以上のフラグの値による条件JMPが用意されている．これらを表2·2に示す．これらの条件JMPは通常，CMP命令によって2つの値を比較した後に書く．符号なし整数の比較の場合は，"Above(より大きい)," "Below(より小さい)," "Equal(等しい)"を表す，"A," "B," "E," それとその否定を表す"N"を"J"の後に組み合わせて命令のニーモニックを作る．また，符号付き整数の比較の場合は，"Greater(より大きい)," "Less(より小さい)," "Equal(等しい)"を表す，"G","L","E"と否定を表す"N"を"J"の後に組み合わせて命令のニーモニックを作る．

プログラムでは，ある処理を何回か繰り返し実行することがよくある．このような繰り返し処理は上に述べたCMP命令と条件JMPを使用してももちろん記述できるが，図2·16に示すようにLOOP命令を使用してもよい．LOOP命令はCXから1を減算する処理を実行し，ZFが1であればオペランドに示したラベルへジャンプする．したがって，図2·16のようにラベルLABEL1からLOOP命令までの処理をCXに初期設定した回数だけ繰り返し実行する．LOOP命令で注意すべきことは，ZFを調べる前にCXから1を減算することである．したがって，CXの初期値が0の場合は，繰り返し回数は0回ではなく，65536回になる．CXの値が0であるのに対して繰り返し回数を0回にしたければ，JCXZ

表 2・1　単一フラグによる条件 JMP

フラグビット	条件 JMP 命令	命令の形式	ジャンプする条件
CF	JC	JXXX ラベル	CF＝1
	JNC		CF＝0
PF	JP/JPE		PF＝1
	JNP/JPO		PF＝0
ZF	JZ/JE		ZF＝1
	JNZ/JNE		ZF＝0
SF	JS		SF＝1
	JNS		SF＝0
OF	JO		OF＝1
	JNO		OF＝0

表 2・2　複数フラグによる条件 JMP

CMP X,Y
条件 JMP 命令　ラベル

	条件 JMP 命令	ジャンプする条件
符号なし整数の比較	JA/JNBE	X＞Y
	JAE/JNB	X＞＝Y
	JE/JZ	X＝Y
	JBE/JNA	X＜＝Y
	JB/JNAE	X＜Y
符号付き整数の比較	JG/JNLE	X＞Y
	JGE/JNL	X＞＝Y
	JE/JZ	X＝Y
	JLE/JNG	X＜＝Y
	JL/JNGE	X＜Y

図 2・16　LOOP 命令と JCXZ 命令

命令を使用して繰り返し処理をスキップするように書くことができる．ここで，JCXZ 命令は，CX が 0 のときオペランドに指定したラベルへジャンプする条件 JMP である．

また，以上述べてきたすべての条件 JMP と LOOP 命令は，short JMP と同様に次の命令のアドレスを中心にして，－128～＋127 バイトの範囲にしかジャンプできない．もし，条件によって short JMP 以上の距離に制御を移行したい場合は，JMP 命令と組み合わせて 2 回ジャンプを作るようにすればよい．

ジャンプ命令

ジャンプ命令のコードは，2 バイト，3 バイト，5 バイトの 3 種類があり，それぞれショートジャンプ，ニアージャンプ，ファージャンプ（ロングジャンプと書いている本もある）と呼ぶ．

本文で述べたように，近くに制御を移行させる場合にはジャンプ命令のコードを短くできるが，ユーザがジャンプ命令を書くアドレスと制御の移行先の距離を予想して，採用するジャンプ命令の種類を決めることはプログラミングにおいて適当ではない．インテルのアセンブラ ASM 86 および ASM 286 では

```
JMP  制御移行先のラベル
```

のように書くだけでよい．ジャンプ命令の種類はアセンブラが自動的に最適なものを選択してくれる．しかし，ラベルが前方参照の場合のようにアセンブラが自動的にその種類を決定できないとき

```
JMP SHORT ラベル
JMP NEAR PTR ラベル
JMP FAR PTR ラベル
```

のようにユーザが明示することもできる．

2-4 ストリング命令

〔1〕 **ストリング命令の動作と種類**　プログラムにおいて，メモリに連続して定義されるデータを扱うことが多い．たとえば，メモリの8000H番地から始まる100Hバイトのデータを，メモリの9000H番地から90FFH番地までの領域に転送するいわゆるブロック転送がある．80286では，このようなメモリブロックを扱う上で有効なストリング命令と呼ばれるいくつかの命令が用意されている．表2·3にストリング命令の一覧を示す．ストリング命令のオペコードニーモニックは最後にSと，バイト単位の転送かワード単位の転送かによって

表 2·3　ストリング命令

オペコードニーモニック	動　作
MOVSB MOVSW	DS:SI で指定されるメモリの1バイトまたは1ワードの値を ES:DI で指定されるメモリに転送してから SI, DI を1または2だけ増加または減少させる.
CMPSB CMPSW	DS:SI で指定されるメモリの1バイトまたは1ワードの値から ES:DI で指定されるメモリの1バイトまたは1ワードの値を減算してから、SI, DI を1または2だけ増加または減少させる．このときメモリの値は変化せず FLAG の値が変化するだけである.
SCASB SCASW	AL または AX から ES:DI で指定されるメモリの1バイトまたは1ワードの値を減算してから、DI を1または2だけ増加または減少させる．このとき AL, AX の値は変化せず，FLAG の値が変化するだけである.
LODSB LODSW	DS:SI で指定されるメモリの1バイトまたは1ワードの値を AL, AX に代入してから，SI を1または2だけ増加または減少させる.
STOSB STOSW	AL または AX の値を ES:DI で指定されるメモリの1バイトまたは1ワードに代入してから，DI を1または2だけ増加または減少させる.
INSB INSW	DX で指定される I/O アドレスの1バイトまたは1ワードの値を ES:DI で指定されるメモリに転送してから，DI を1または2だけ増加または減少させる.
OUTSB OUTSW	DS:SI で指定されるメモリの1バイトまたはワードの値を DX で指定される I/O に出力してから，SI を1または2だけ増加または減少させる.

ここで，オペコードニーモニックがBで終わっているときはバイト転送であり，アキュムレータとしてALを使用し，インデックスレジスタの増加，減少分は1である．また，オペコードニーモニックがWで終っているときはワード転送であり，アキュムレータとしてAXを使用し，インデックスレジスタの増加，減少分は2である．

FLAGのDFが0のときインデックスレジスタを増加させる．FLAGのDFが1のときインデックスレジスタを減少させる．DSはセグメントオーバライドプリフィックスによって，任意のセグメントレジスタに変更することができる．

BまたはWが付く．処理の対象は命令によって暗黙的に決まっているので，オペランドを指定する必要はない．

MOVS命令は，図2·17に示すようにメモリのDS:SIで指定される1バイトまたは1ワードの値を，ES:DIで指定されるメモリに転送する．MOVSB命令のときは1バイトの転送であり，MOVSW命令のときは1ワードの転送となる．また，このときSI，DIのインデックスレジスタを自動的に更新する．MOVSB命令のときは，インデックスレジスタを1だけ更新し，MOVSW命令のときは，インデックスレジスタを2だけ更新する．さらに，インデックスレジスタの更新は，FLAGのDFによって増加か減少かが制御される．インデックスレジスタは，DF＝0のとき増加され，DF＝1のとき減少される．

図 2・17 MOVS命令によるデータ転送

表 2・4 DFの設定

オペコードニーモニック	動作
CLD	DFを0にクリアする．
STD	DFを1にセットする．

表2·3に示すMOVS命令以外のストリング命令においても同様の動作を実行する．ストリング命令のオペランドは，ALまたはAX，メモリのバイトまたはワードのデータ，そしてアドレスがDXで指定されるI/Oであり，ストリング

```
SOURCE_PTR DD 00008000H
DESTINATION_PTR DD 00009000H
          ⋮
LDS SI,SOURCE_PTR
  ➡ 変数SOURCE_PTRの上位ワードをDSに代
    入し，下位ワードをSIに代入する
LES DI,DESTINATION_PTR
  ➡ 変数DESTINATION_PTRの上位ワードを
    ESに代入し，下位ワードをDIに代入する
CLD ➡ DFを0にクリアする
MOVSW
```

図 2・18 MOVSW命令を使用した1要素のデータ転送

表 2・5 REP プリフィックスの種類

REP プリフィックスの使用	動　作
REP [ストリング命令]	REP の直後に指定されたストリング命令を、CX で定義された回数だけ繰り返し実行する．
REPE [ストリング命令] (REPZ)	REPE の直後に指定されたストリング命令を、CX で定義された回数だけ繰り返し実行する．ただし、ZF＝0 の条件で、CX の値に関係なく、繰り返しを終了する．
REPNE [ストリング命令] (REPNZ)	REPNE の直後に指定されたストリング命令を、CX で定義された回数だけ繰り返し実行する．ただし、ZF＝1 の条件で、CX の値に関係なく、繰り返しを終了する．

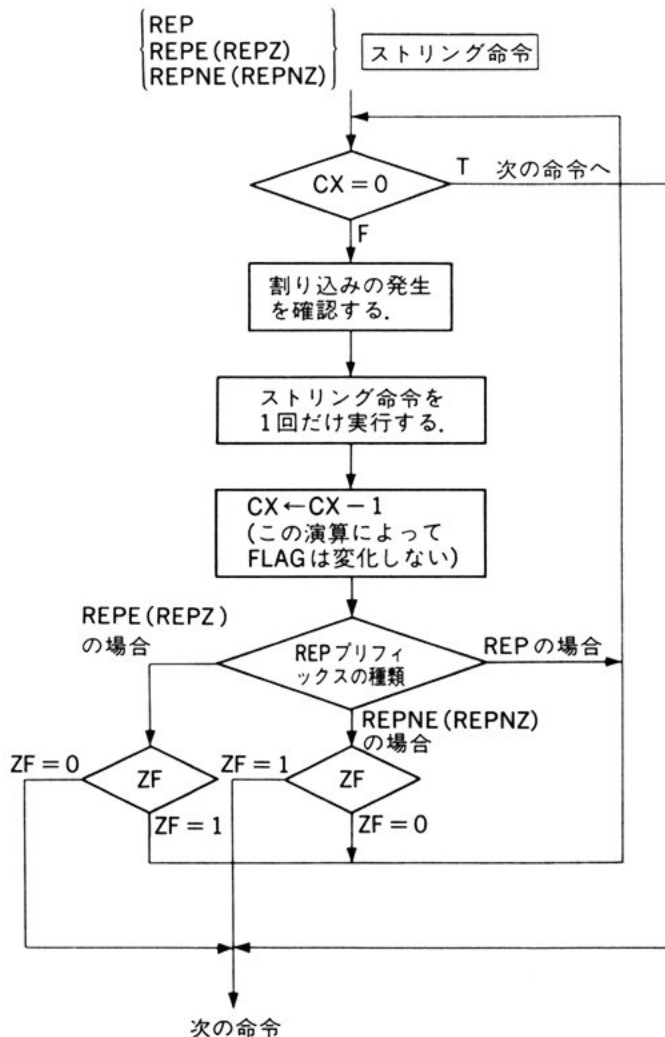



図 2・19 REP プリフィックス付きのストリング命令

命令の種類によって2つの処理対象が暗黙的に定義されている．ここで，ストリング命令の処理対象がメモリのとき，データが代入されるメモリは必ずES:DIによって指定される．逆にソースデータをもつメモリはDS:SIによって指定される．ソースメモリを指定するセグメントレジスタは，一般にDSであるが，セグメントオーバライドプリフィックスを指定することによって，他の任意のセグメントレジスタに変更することができる．ESを変更することはできない．インデックスレジスタのSI (source index)，DI (destination index) の名前はストリング命令における上のような使い方から与えられたものである．ストリング命令以外の命令においては，インデックスレジスタとしてSI，DIのどちらを使用してもかまわない．

また，ストリング命令に関係するインデックスレジスタは自動的に更新される．MOVS命令，CMPS命令においてはSI，DIの両方が，またSCAS命令，STOS命令，INS命令においてはDIが，LODS命令，OUTS命令においてはSIがそれぞれ，DFの制御によって増加または減少される．DF＝0のとき，バイト操作のストリング命令では1だけ，またワード操作のストリング命令では2だけ増加される．DF＝1のときは，同様にして減少される．

ストリング命令は，以上のように暗黙的に使用するいくつかのレジスタまたは制御フラグをもつので，ストリング命令を実行する前にそれらを初期設定する必要がある．ここで，FLAGのDFの1ビットだけの初期設定のために，表2·4に示すようなCLD命令，STD命令が用意されている．たとえば，メモリの8000H番地にある1ワードのデータを，メモリの9000H番地に転送するためにはMOVSW命令を用いて，図2·18に示すようにプログラムを書く．

〔2〕 **REPプリフィックス付きストリング命令**　しかし，図2·18に示すようにストリング命令だけではデータブロックの1要素しか扱うことができない．表2·5に示すREPプリフィックスをストリング命令の前に書くことによって，ストリング命令を図2·19に示すように繰り返し実行することができる．ここで，実行の繰り返し回数はCXによって定義される．また，REPE (REPZと書いてもよい) プリフィックスはCXが0にならなくても，ZFが0になればストリング命令の繰り返しを終了する．逆にREPNE (REPNZと書いてもよい) の場合はZFが1になれば，CX≠0の状態であっても繰り返しを終了する．REPE，REPNEは，CMPS命令，SCAS命令とともに用いる．したがって，メモリの

8000H 番地から定義された 100H ワードのデータを，9000H 番地から始まるメモリにワード単位で転送する処理は図 2·20 に示すようになる．なお，ストリング命令の繰り返し処理には REP プリフィックスを付ける代わりに LOOP 命令を使用してもかまわない．

```
SOURCE_PTR DD 00008000H
DESTINATION_PTR DD 00009000H
            ⋮
LDS SI,SOURCE_PTR
LES DI,DESTINATION_PTR
MOV CX,100H
CLD
REP MOVSW
```

図 2・20　MOVSW 命令を使用した 100H ワードのデータ転送

セグメントオーバライド(インストラクション)プリフィックス

8086 が論理アドレスから物理アドレスを計算するとき，セグメントレジスタの値をベースアドレスとして使用するが，CS, DS, SS, ES のどのレジスタを使用するかは暗黙的に決まっている．しかし，命令の前に

2EH ➡ 続く命令が CS を使用してメモリ参照をする．
3EH ➡ 続く命令が DS を使用してメモリ参照をする．
36H ➡ 続く命令が SS を使用してメモリ参照をする．
26H ➡ 続く命令が ES を使用してメモリ参照をする．

の 4 種類の 1 バイトのセグメントオーバライドプリフィックスを指定することによって，使用するセグメントレジスタをユーザが任意に明示することができる．

プログラムでセグメントオーバライドプリフィックスを表現するとき

```
MOV AX,ES:WORD PTR 100H
```

のように，メモリオペランドの前に CS:, DS:, SS:, ES: のように書く．しかし，セグメントレジスタの種類を考えながらプログラムを書くことは煩わしいので，ASSUME 宣言文という疑似命令が用意されている．ASSUME 宣言を書くことによってセグメントオーバライドプリフィックスの使用を統一的にアセンブラにまかせることができる．

2-5 拡張命令

C 言語の関数，PASCAL 言語および PL/M 言語などの手続きのような，高級言語の手続きについて考えるとき，手続き内で使用する変数をデータセグメントのようなスタティック（静的）な領域に定義する場合と，手続きが引用されている間だけスタックセグメントにダイナミック（動的）に定義する場合がある．静的な変数はプログラムが実行されている間，常にメモリ上に存在するが，動的な変数は部分的な手続きを実行する間だけ，スタックセグメントに一時的に存在するだけである．C 言語の場合，関数で定義するスタティック変数は前者の例であり，オート変数が後者の例である．

80286 は，スタックセグメントに動的な変数領域を割り付ける ENTER 命令と，ENTER 命令によって作成した動的変数領域をスタックから削除する LEAVE 命令をもつ．図 2·21 に ENTER 命令とその動作を示す．ENTER 命令は 2 つの定数オペランドをもち，一般に手続き定義の先頭に書く命令である．左側のオペランドによって動的変数領域のバイト数を指定する．右側のオペランドは手続きのネストのレベルを表す．C 言語の関数とか，図 2·22 に示す手続き A のようなもっとも外側の手続きで ENTER 命令を書くときは，レベルを 1 に設定すればよい．

たとえば，スタックセグメントに 100 バイトの動的変数領域を作るとき

```
ENTER 100,1
```

の命令を手続きの先頭に書けばよい．この命令を実行したとき，図 2·23 に示すような領域がスタックセグメントに定義される．ENTER 命令がスタックセグメントに定義する領域をスタックフレームと呼ぶ．スタックフレームは動的変数領域とディスプレイ領域からなるが，レベルを 1 に指定した場合は ENTER 命令実行前の BP の値とスタックフレームのベースを示す BP の値 BP_A が保存されている．また，スタックフレームのベースアドレスは，BP によって指定されるからそれぞれの動的変数は

```
WORD PTR [BP-4]
BYTE PTR [BP-6]
```

のように参照することができる．

ここで再び図 2·22 において，手続き B について考える．手続き B において

```
ENTER DYNAMIC,LEVEL
```

‖ 等 価

```
LEVEL ← LEVEL MOD 32
PUSH BP
TEMP ← SP
if LEVEL > 0

then repeat (LEVEL-1)
     BP ← BP-2
     PUSH WORD PTR [BP]
     endrepeat
     PUSH TEMP
end if

BP ← TEMP
SP ← SP-DYNAMIC
```

ここで，repeat は repeat（n）の次から endrepeat までの処理を n 回繰り返すものとする．また，TEMP は内部作業のための変数である．

LEVEL MOD 32 は LEVEL を 32 で割った余りを求めることを意味する．

DYNAMIC は，動的変数領域のバイト数を表す 1 ワードの定数であり，LEVEL は手続きのネストを表す 1 バイトの定数である．

図 2・21 ENTER 命令の動作

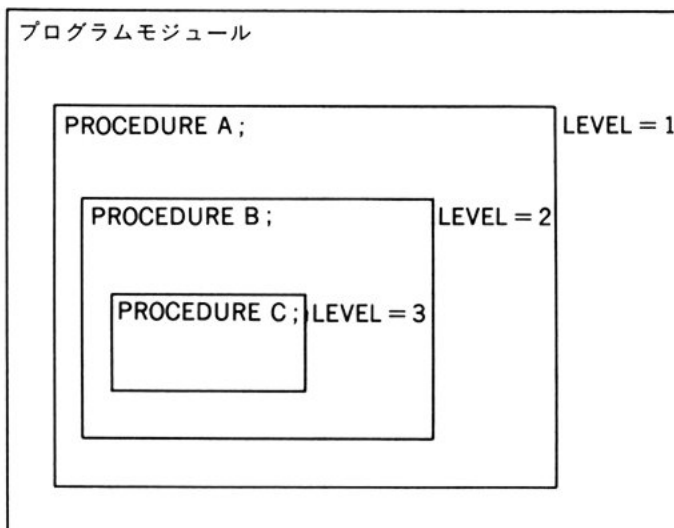



図 2・22 ネスト構造の手続き定義

は，手続き A で定義した変数も参照できなくてはならない．そのために，手続き B のスタックフレームに，手続き A のスタックフレームの BP の値 BP_A を記録しておくことが有効になる．したがって，手続き B で 14 バイトの動的変数領域を定義するときは，レベルを 2 にして

```
ENTER 14,2
```

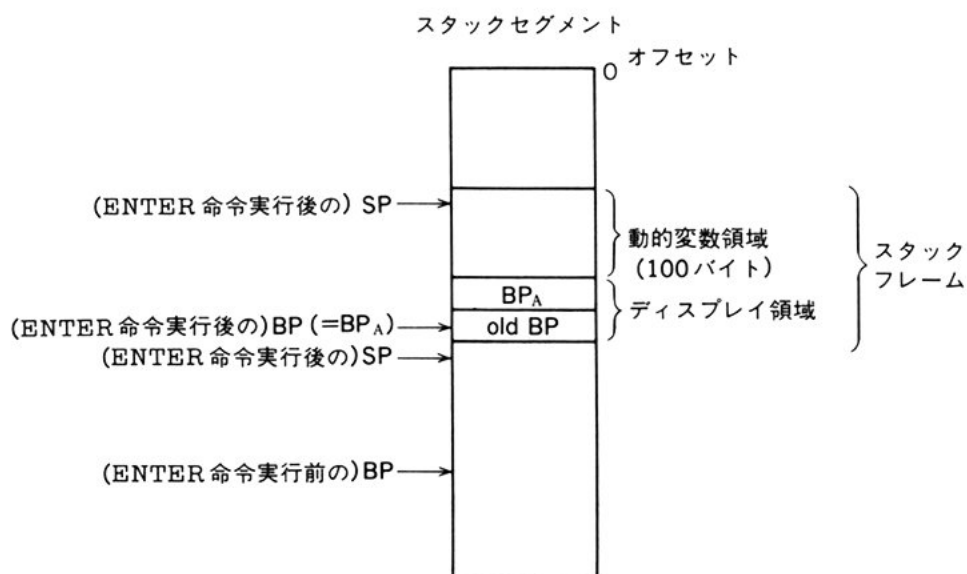



図 2・23　ENTER 命令によって作られるスタックフレーム

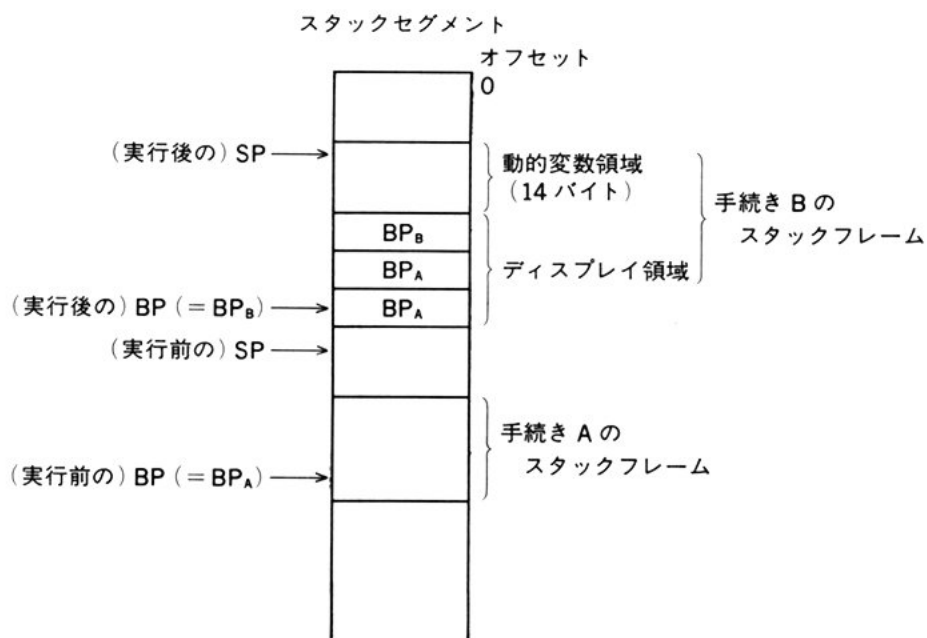



図 2・24　ENTER　14，2 の実行

のように書く．このとき，図2·21に示したENTER命令の動作からわかるように，図2·24に示すようなスタックフレームが作られる．

さらに，図2·22に示した手続きCで実行するENTER命令は，レベルを3にすれば，そのスタックフレームのディスプレイ領域に，手続きA，手続きBの

```
LEAVE
```

Ⅲ 等 価

```
MOV   SP,BP
POP   BP
```

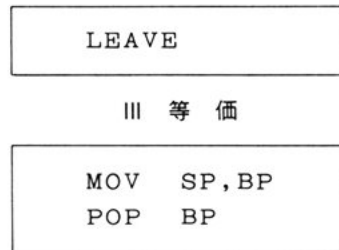



図 2·25 LEAVE命令の動作

```
I EQU WORD PTR [BP-2]
J EQU WORD PTR [BP-4]
K EQU WORD PTR [BP-6]

PROC_A PROC

      ENTER 6,0

      IN AX,0
      MOV I,AX
      IN AX,2
      MOV J,AX

      ADD AX,I
      MOV K,AX

      OUT 4,AX

      LEAVE
      RET

PROC_A ENDP
```

注）EQUはEQUの右側の定義に，左側に示すようなシンボルを定義する疑似命令である．なお，この手続きの処理自体には意味はあまりない．

図 2·26 ENTER命令，LEAVE命令の使用

スタックフレームの BP の値 BP_A，BP_B を記録することができる．

もちろん，高級言語のすべてが，ENTER 命令のレベルを 1 以上にするような処理を必要としているわけではない．PL/M 言語の動的変数を作るリエントラントな手続きはネストすることが禁じられている．このような場合，レベルを 0 にして

```
ENTER 6,0
```

のように使用してもよい．このとき，ENTER 命令の実行は

```
PUSH BP
MOV BP,SP
SUB SP,6
```

の処理と同じである．

ENTER 命令で作成したスタックフレームは，図 2·25 に示す LEAVE 命令を実行して削除することができる．LEAVE 命令はオペランドをもたず，RET 命令の上に書けばよい．このように，ENTER 命令，LEAVE 命令は常に対で使用し，たとえば 3 ワードの動的変数を使用する手続きの定義は，レベルを 0 にするとアセンブリ言語で図 2·26 のように書くことができる．

3. プロテクトモードでの使用

80286の本来の能力はプロテクトモードにおいて発揮される．リアルモードの場合とプロテクトモードの場合では，メモリ管理の方法がまったく異なることを本章で学ぶ．それでもなお，応用プログラムを書くプログラマには，8086と80286の違いは見わけられないだろう．リアルモードの命令は，プロテクトモードでもそのまま使用できる．しかし，ポインタを直接扱うプログラムは，リアルモードとプロテクトモードの間で互換性がなくなることに注意しなければならない．本章では，プロテクトモードでのメモリ管理について説明した後，リアルモードからプロテクトモードへ変換するプログラムを示す．

3-1 セグメントキャッシュ

〔1〕 **プロテクトモードの論理アドレス**　プロテクトモードにおいてもメモリのアドレスは，論理アドレスによって表し，論理アドレスはセグメントセレクタとオフセットからなる．オフセットは，リアルモードと同様にメモリ参照の状況によってIP，SPまたは命令のオペランドで指定される．また，セグメントセレクタもCS，DS，ES，SSの4種類のセグメントレジスタに入っている値が使用される．図3·1に再びセグメントレジスタとセグメントキャッシュを示すが，プロテクトモードにおいては，このセグメントキャッシュの扱い方が，リアルモードの場合とは異なる．

リミットフィールドには，リアルモードの場合は常に，OFFFFHの値が設定されたのに対して，プロテクトモードの場合は，1からOFFFFHまでの任意の最大オフセット値を設定することができる．リアルモードの場合のベースアドレスは，セグメントセレクタを4ビットだけ左シフトした値（すなわち16倍した値）が設定されたが，プロテクトモードでは，ベースアドレスの値は任意の24ビットの値を定義することができる．ここで，セグメントセレクタは，セグメントのベースアドレスとは直接に関係のない，セグメントを論理的に識別するためのインデックスとなる．さらに，プロテクトモードでは，アクセスライト（AR）フィールドの1バイトの値によって，図3·2のような形式で，セグメントの属性を定義することができる．

以上のように，セグメントキャッシュによって表される6バイトのデータを**セ**

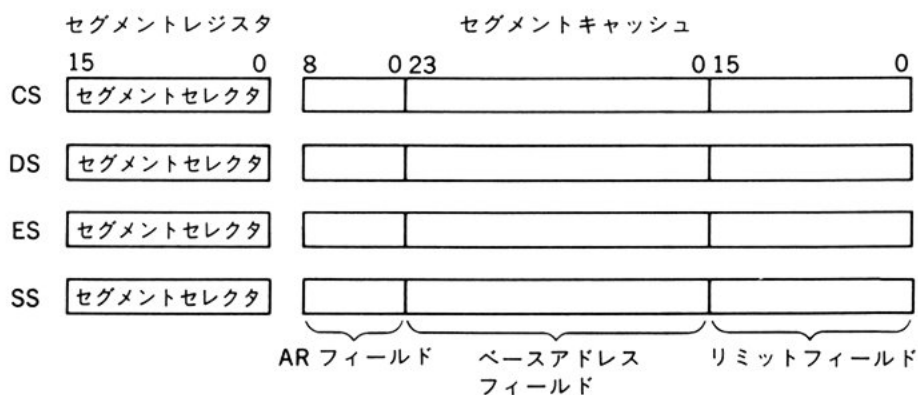



図 3·1　セグメントレジスタとセグメントキャッシュ

グメントディスクリプタと呼ぶ．80286は，メモリを参照するときセグメントディスクリプタを使用して，オフセットが定義されたセグメントリミットを超えていないかどうか，メモリ参照の方式がアクセスライトに定義した規則にかなっているかどうか，などを検査する．もし，それらの検査の中で1つでも，セグメントディスクリプタに定義した規則に違反していれば，メモリ保護のために80286はその命令を中断して，例外割り込みを発生する．命令の実行がすべての検査をパ

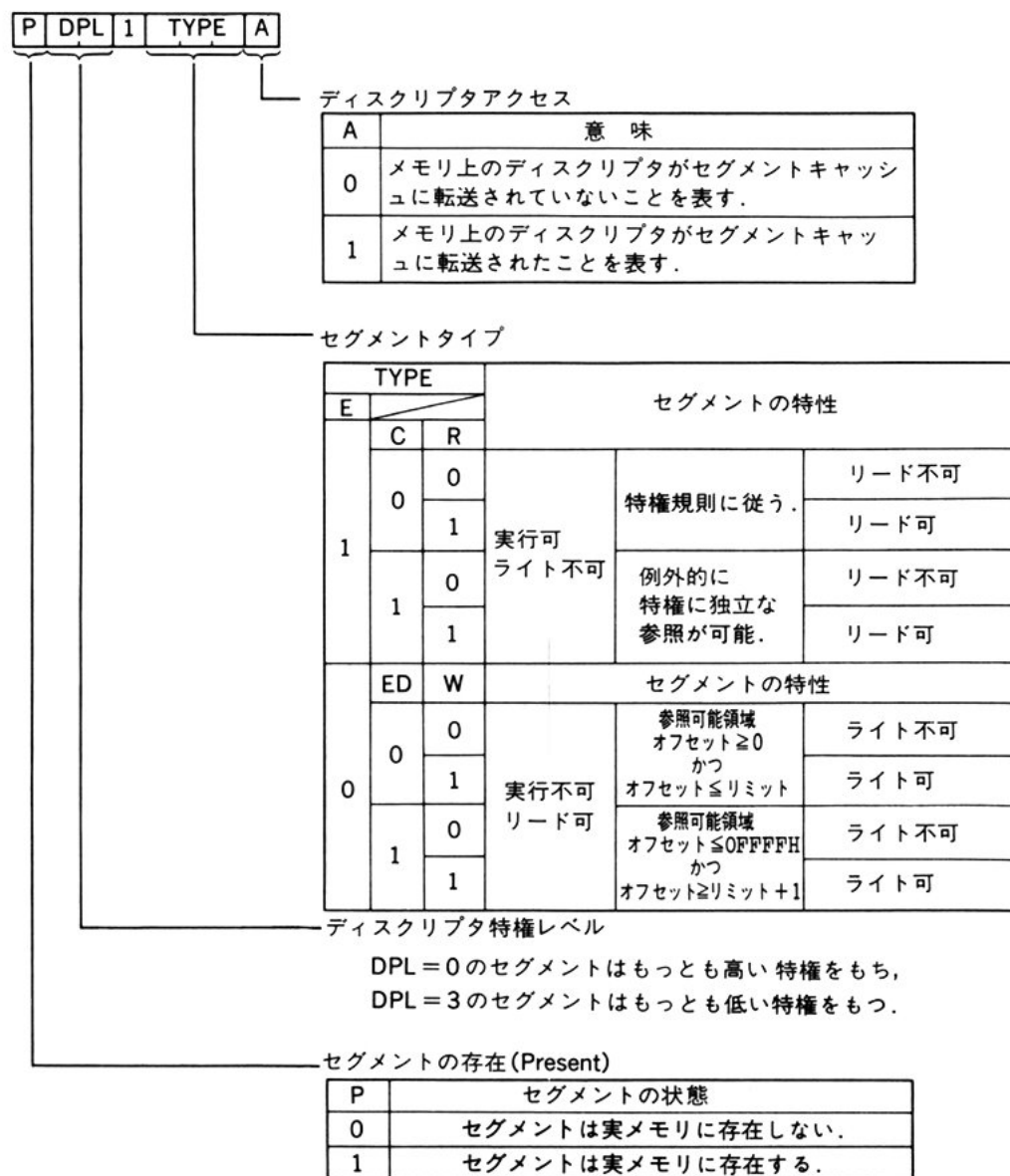



図 3・2 セグメントキャッシュのアクセスバイト

スしたときにだけ，セグメントキャッシュのベースアドレスにオフセットを加えた物理アドレスをアドレスバスに出力し，バスサイクルを実行する．

セグメントキャッシュに定義するセグメントディスクリプタは，80286がメモリ参照の正当性を評価するための重要な評価基準である．したがって，誤まったセグメントディスクリプタがセグメントキャッシュに定義されたのでは何にもならない．したがって，セグメントキャッシュには直接に値を代入したり，また逆に，読み出したりできないようになっている．セグメントキャッシュの値は，80286がその正当性を検査してから設定される．それでは，どのようにしてセグメントキャッシュの値が決まるのであろうか．

プロテクトモードの学習

80286のプロテクトモードは，マルチタスクのオペレーティングシステムを設計するうえで基本的な機能を，CPUのハードウェアまたはマイクロコードに広く応用できる形でもたせたということができる．このような80286の各機能を学習する場合，とにかく80286をプロテクトモードにして多くのテストプログラムを実行させてみるのが一番よい．しかし，初心者にとって困難なことは，80286のプロテクトモードの動作をひととおり知っていなければ，リアルモードからプロテクトモードに切り換えて80286をうまく動作させることが難しい要素があることである．この問題は，リアルモード，プロテクトモードの両方で利用できるモニタプログラムを提供し，初心者にとって最初は困難な処理をモニタプログラムの中に作り込んでおくことによって解決できる．もっといいのは80286と同等の機能をソフトウェアで実現したシミュレータを準備することである．このようにすれば80286のマシンを持たない人でも80286の学習をすることができる．

インテルから80286のモニタSDM 286およびシミュレータSIM 286が提供されているが，現在では80286を採用した多くのパーソナルコンピュータが各社から出ているのだから，ビジネス用のOSとは別にプロテクトモードの機能をユーザがテストできるようなモニタなども提供されていいような気がするのだが…．

3-2 ディスクリプタテーブル

〔1〕 **ディスクリプタテーブル**　セグメントレジスタは，最も最近に使用されたセグメントを表すと考えることができる．新しいセグメントを参照する場合，4 種類のセグメントレジスタのうち，適当なレジスタに目的のセグメントを選択するセグメントセレクタを初期設定する．リアルモードのセグメントキャッシュにおいて，実質的に意味のあるデータはベースアドレスだけであり，他のアクセスライト，リミットの値は固定である．そして，セグメントセレクタとベースアドレスは

ベースアドレス＝セグメントセレクタ×16

の関係をもち，セグメントの識別はセグメントセレクタで表現されたベースアドレスによって行った．これに対してプロテクトモードでは，アクセスバイト，ベースアドレス，リミットのセグメントディスクリプタの 3 つの要素によってセグメントを柔軟に定義できる．そのために，OS，ユーザプログラムを含めたシステムで使用するすべてのセグメントに対するセグメントディスクリプタを管理しな

図 3・3　GDT と LDT

図 3・4 セグメントディスクリプタ

GDT

セグメントディスクリプタ

LDT

セグメントディスクリプタ

セグメントレジスタにセレクタを代入するとき自動的にセグメントディスクリプタの下位6バイトの値が対応するセグメントキャッシュに代入される.

15 3 2 0

セグメントレジスタ

セグメントキャッシュ

TI = 0 GDT, 1 LDT

ディスクリプタインデックス

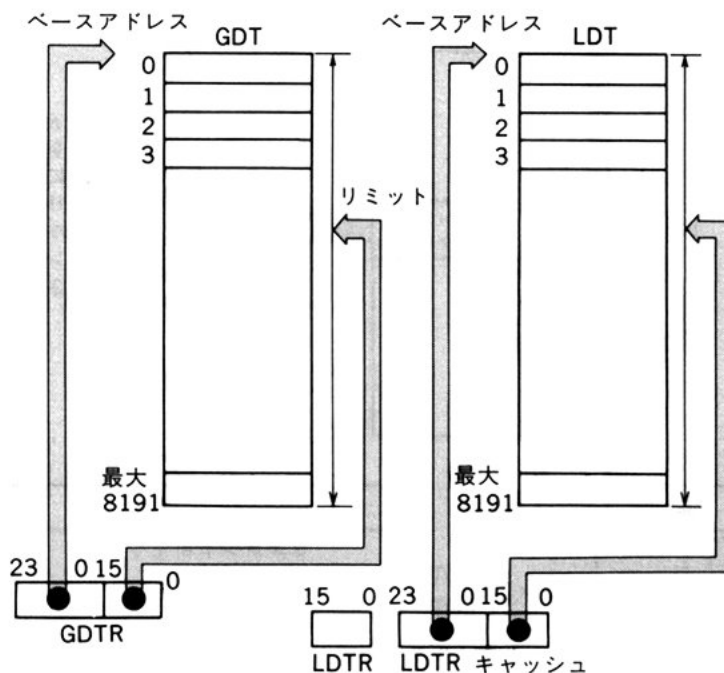
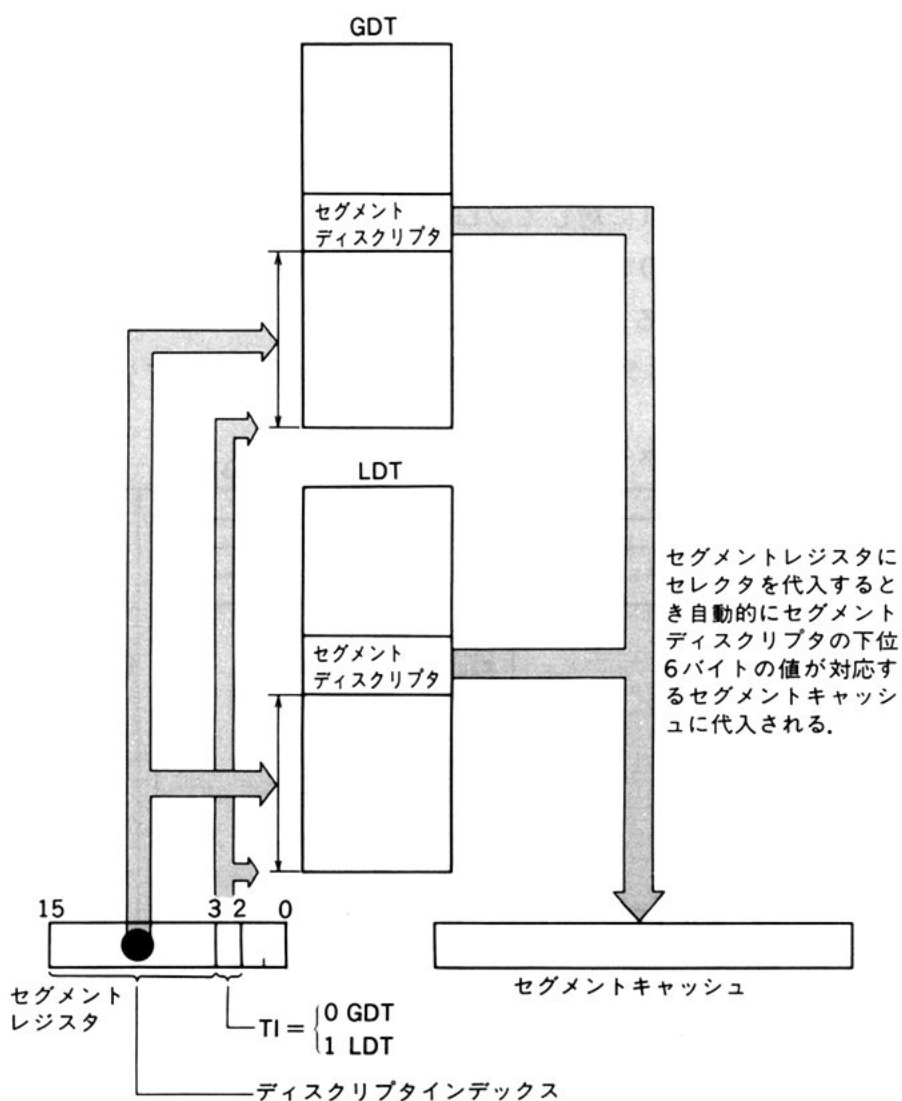



図 3・5 セグメントレジスタとセグメントキャッシュへの代入

ければならない．

80286は，図3·3に示すような**GDT**（**グローバルディスクリプタテーブル**），**LDT**（**ローカルディスクリプタテーブル**）と呼ばれるメモリに定義された2つのテーブルによって，ディスクリプタを管理することができる．GDT，LDTはともに，最大が64Kバイトの大きさをもち，図3·4に示す形式のセグメントディスクリプタをそれぞれ最大8191まで定義することができる（ディスクリプタテーブルのスロット0は使用できないので8192ではない）．80286のセグメントディスクリプタは，6バイトの大きさであるが，ディスクリプタテーブルに定義するときは，図3·4に示すように8バイトを定義する．8バイトのセグメントディスクリプタの上位2バイトは，32ビットのマイクロプロセッサ**80386**のディスクリプタと互換性を保つために0を代入しておく（80286プロテクトモードのプログラムは，80386でもそのまま走らせることができる）．

〔2〕 **セグメントセレクタとディスクリプタテーブル**　このように，プロテクトモードでは，すべてのセグメントに対応するセグメントディスクリプタを，GDT，LDTのどちらかに定義しておかなければならない．そして，プロテクトモードのセグメントセレクタは，ディスクリプタテーブルの中から，1つのディスクリプタを選択するために使用される．図3·5に示すように，セグメントセレクタのビット2（TI，テーブルインジケータ）がGDTかLDTの識別に使用される．TIが0のとき，GDTが選択され，TIが1のとき，LDTが選択される．セグメントセレクタの上位13ビットが，ディスクリプタテーブルのインデックス（スロット番号）を指定する．すなわち，プロテクトモードでは，セグメントセレクタによって，1つのセグメントディスクリプタが決まり，そのセグメントディスクリプタのデータによって，セグメントのベースアドレス，大きさ，その他の属性が決まる．このセグメントセレクタをセグメントレジスタに代入するとき，自動的に対応するセグメントキャッシュに，セグメントディスクリプタの下位6バイトが代入される．

〔3〕 **ディスクリプタテーブルの定義**　したがって，80286をプロテクトモードで動作させるためには，プロテクトモードに移る前に，GDT，LDT（少なくともGDT）にセグメントディスクリプタの初期設定をしなければならない．GDTのベースアドレスとリミットは，図3·6に示すようなGDTRによって定義できる．

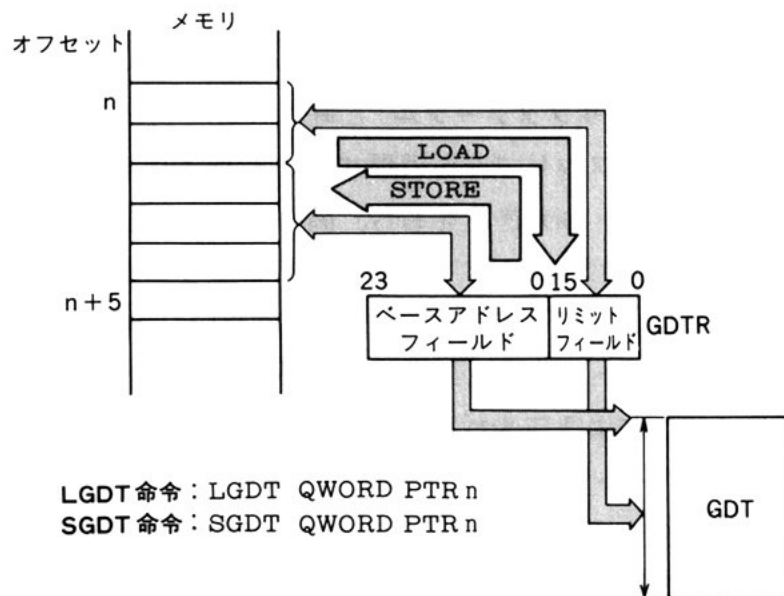



図 3・6 GDTの定義

図 3・7 LDTRとLDTRキャッシュ

15 ... 8	7 ... 0	
リミット		0
ベースアドレス A_0-A_{15}		2
P | DPL | 0 | 0 0 1 0	ベースアドレス A_{16}-A_{23}	4
80386のための予約領域. 80286では0にする.		6

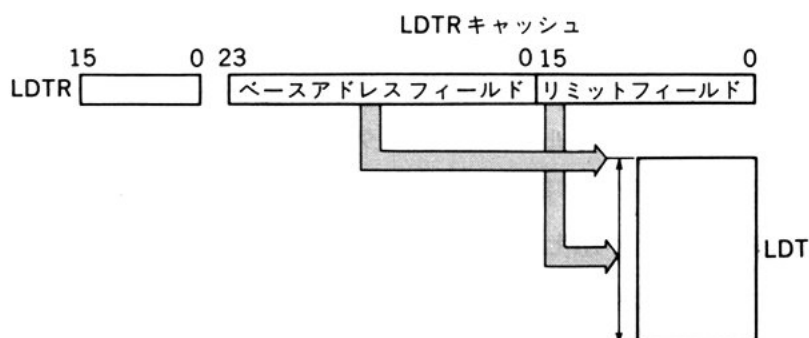



図 3・8 LDTディスクリプタ

GDTRは，40ビットのレジスタで，16ビットのリミットフィールドと24ビットのベースアドレスフィールドからなる．GDTRの値の設定，読み出しは，図3·6に示すLGDT命令，SGDT命令を使用して実行できる．LGDT命令はメモリに定義した6バイトのデータをGDTRに代入する命令である．このとき，GDTRは5バイトの大きさであるから，メモリの6バイトのデータの最上位の1バイトは無視される．また，SGDT命令を使用して，GDTRの内容をメモリに書くこともできる．プロテクトモードに移る前に，少なくともGDTを定義しなければならないので，LGDT命令，SGDT命令はともに，リアルモードでも使用することができる．

一方，LDTのベースアドレスとリミットは，図3·7に示すようにLDTRによって定義されるが，LDTRはGDTRとは形式が異なり，むしろセグメントレジスタのように，キャッシュレジスタをもつ．LDTRキャッシュの大きさは40ビットで16ビットのリミットフィールドと23ビットのベースアドレスフィールドから構成される．

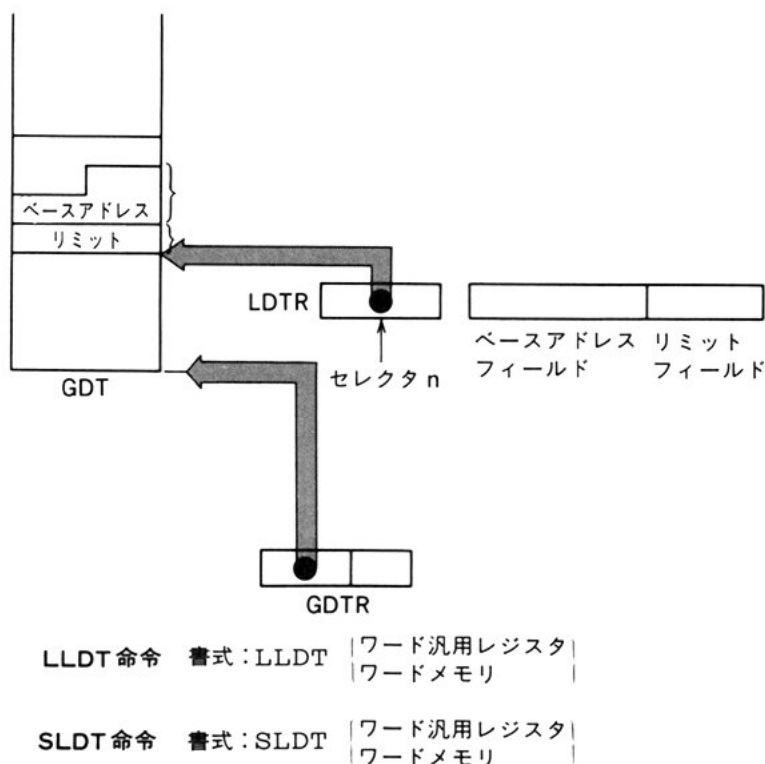



図 3·9　LLDT命令とLDTRキャッシュの設定

3 プロテクトモードでの使用

セグメントキャッシュに代入するデータをセグメントディスクリプタと呼び，ディスクリプタテーブルに定義したのと同じように，LDTRキャッシュに代入するデータを**LDTディスクリプタ**と呼び，図3·8に示すような形式をもつ．LDTディスクリプタの下位5バイトがLDTキャッシュに代入される値である．LDTディスクリプタのアクセスライトは，LDTRキャッシュには代入されない．LDTディスクリプタは，必ずGDTに設定しておく．

そこで，LDTRにセレクタを代入するためには，図3·9に示すように，LLDT命令を使用する．オペランドに指定したワード汎用レジスタ，またはワードメモリから，LDTディスクリプタを選択するセレクタをLDTRに代入すると，自動的にLDTディスクリプタの下位5バイトがLDTRキャッシュに代入される．逆に，SLDT命令を使用して，LDTRの値をオペランドに指定したワード汎用レジスタ，ワードメモリに代入することもできる．このときは，LDTRキャッシュの値は変化しない．また，LDTRキャッシュの値を直接に読むことはできない．なお，LLDT命令，SLDT命令はプロテクトモードにおいてのみ使用可能である．

このように，GDTRとLDTRのレジスタの形式が異なるのは，GDTRは一度定義してしまえば後に変更することはほとんどないのに対して，LDTRはシステムの状況に応じて，変更することがあるためである．

3-3 セグメントレジスタの保護

セグメントレジスタにセグメントセレクタを代入することによって，GDT または LDT からセグメントキャッシュにディスクリプタが代入される．セグメントレジスタに値を代入する命令をまとめると表 3·1 のようになる．しかし，80286 が実行する高機能なメモリ管理は，すべてセグメントキャッシュのディスクリプタを評価基準として行われるからセグメントキャッシュに誤まった値が代入されては意味がない．したがって，80286 は，セグメントレジスタ，セグメントキャッシュの値を変更するとき，代入する値の正当性を検査するようになっている．

〔1〕 **DS，ES の保護**　　たとえば，図 3·10 に示すような GDT が定義されているときに

```
MOV  AX,101000B      ①
MOV  DS,AX           ②
```

のような操作で，DS にセレクタ 101000B を代入することを考える．②の命令において DS, DS キャッシュに値が代入されるまでに，次のような処理が自動的に実行される．

（1） セグメントセレクタが 0 であるかどうか検査する．もし，AX の値が 0 であれば DS に 0 を代入し，DS キャッシュが不正な値をもつことを意味するマークを内部的に付けてから命令を終了する．0 のセグメントセレクタはヌルセレクタと呼ばれ，どのセグメントをも指し示さないことを意味する．したがって，ディスクリプタテーブルのスロット 0 は使用されない．AX の値が 0 でなければ，次の検査に移る．

（2） AX のセレクタによって，指定されるセグメントディスクリプタがディスクリプタテーブルのリミット内に定義されているかどうか検査する．もし，指定したセグメントディスクリプタがテーブルリミットを超えているときは，処理を中断してタイプ 13 の割り込みに入る．このとき，DS, DS キャッシュの値は変化しない．セグメントディスクリプタがテーブルリミットを超えていなければ，次の検査に移る．

（3） ここで，初めてディスクリプタの値をテーブルから読む．しかし，無条件にそのまま DS キャッシュに代入するのではなく，まずアクセスライトの値を

表 3・1 セグメントレジスタを書き換える命令

セグメントレジスタ	セグメントレジスタにセレクタを代入する命令
DS, ES, SS を書き換える命令	MOV, LDS, LES, POP
CS を書き換える命令	far タイプ JMP, far タイプ CALL 割り込み

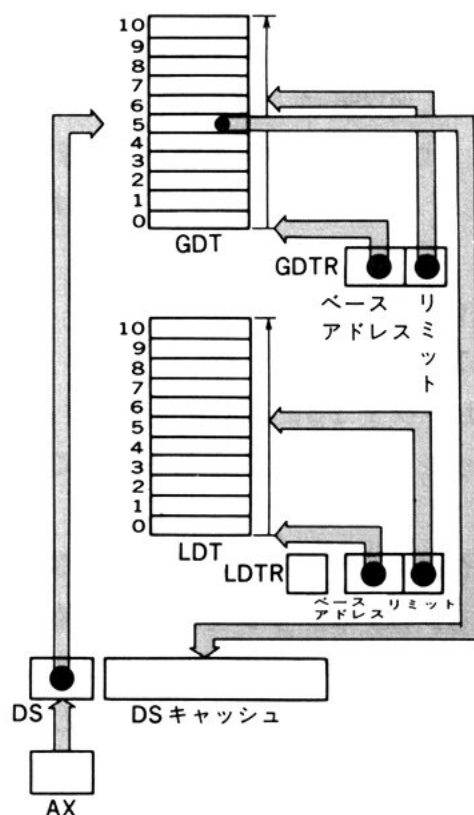



図 3・10 MOV DS, AX の実行

調べる．ここで，このセグメントがデータとして参照することが許されているかどうかを調べる．すなわち，E が 0 であるか 1 であるかである．もし，E が 1 であれば，R が 1 であるかどうかを調べる．もし，このセグメントが，書くことも読むことさえも許されていないならば，80286 は処理を中断して，タイプ 13 の割り込みを発生する．E が 0 であるか，また E が 1 であっても R が 1 であれば，次の検査に移る(図 3・2 参照)．

（4） 最後に，P が 1 であるかどうかを検査する．もし，P が 0 であれば，デ

ィスクリプタは定義されていても，実際のセグメントは実メモリに存在しないことを意味するから，80286は処理を中断して，タイプ11の割り込みを発生する．Pが1であれば次の処理に移る．

（5） 以上の検査をすべてパスした後，80286はDSに101000Bを代入し，対応するGDT(5)に定義されているディスクリプタをDSキャッシュに代入する．また，このとき80286は自動的にGDT(5)に定義されているディスクリプタのAを1にする．

(1)から(5)までの，DSに関する扱いは，ESについてもまったく同様である．

〔2〕 **SSの保護**　次にSSの場合について考える．たとえば

```
MOV  AX,1000000B
MOV  SS,AX
```

の処理によって，SSに1000000Bを代入する場合について考える．

SSもDS，ESと同様にセグメントセレクタを代入することができるが，SSはスタックセグメントを指定するという性格上，SSおよびSSキャッシュに設定できる値はDS，ESの場合と少し異なる部分がある．

（1） SSに代入するセグメントセレクタが0かどうかを検査するのは，DS，ESの場合と同様であるが，セレクタが0のときは，SSに0を代入しないでタイプ13の割り込みを発生する．このとき，SS，SSキャッシュの値は変化しない．スタックセグメントは，CALL命令，RET命令，割り込み，IRET命令において自動的に使用されるから，SSには0を代入して，無効にすることは許されていない．セレクタが0でなければ次の検査に移る．

（2） DS，ESの場合とまったく同様に，セレクタに対応するディスクリプタがディスクリプタテーブルの中にあるかどうかを検査する．

（3） (2)の検査に合格した後，やはりディスクリプタを読み，アクセスライトの値を検査するが，SSに代入する場合はリード，ライトが両方とも可能でなければならない．すなわち，アクセスライトのビットが，E＝0，W＝1でなければならない．もし，そうでなければ，タイプ13の割り込みが発生する．

（4） 最後に，やはりPが1かどうかを検査する．もし，P＝0であればタイプ12の割り込みが発生する．

(1)から(4)の検査をすべてパスして，初めてSSにAXのセレクタが代入され，

SSキャッシュに対応するディスクリプタが代入される．このときも，セグメントキャッシュにディスクリプタが代入されたことを表すため，80286は自動的にメモリのディスクリプタのAに1を書く．

〔3〕 **CSの保護**　CSはfar CALL，far JMP，割り込みによって，IPと同時に変更される．たとえば

```
JPTR DD 680000H
        ⋮
JMP JPTR
```

の処理は間接JMPを使用して，セグメントが68H，オフセットが0の命令に制御を変えるもので，このとき，IPに0が代入されるのと同時に，CSに68Hが新しく代入される．この場合も，CSに代入されるセレクタと対応するディスクリプタは次のように検査される．

（1） CSに代入されるセレクタが0かどうかを検査する．もし，セレクタが0であれば処理を中断し，タイプ13の割り込みに入る．CSの場合もSSのときと同様に80286の実行において，CSを0にして無効にするような状況はありえないから，CSに0を代入することは許されていない．

（2） セレクタに対応するディスクリプタが，ディスクリプタテーブルのリミット内に定義されているかどうかを検査する．もし，対応するディスクリプタがリミットを超えているものであればタイプ13の割り込みを発生する．

（3） 上述の検査に合格したならばディスクリプタを読み，アクセスライトを調べ実行可能かどうかを検査する．もし，Eが0で実行できないデータが定義されているセグメントである場合，タイプ13の割り込みを発生する．

表3·2 セグメントタイプの検査

セグメントレジスタ ＼ アクセスライト	S(ARのビット4)＝1(セグメントディスクリプタであること)			
	E＝0(データセグメント)		E＝1(コードセグメント)	
	W＝0(ライト不可)	W＝1(ライト可)	R＝0(リード不可)	R＝1(リード可)
DS	○	○	×	○
ES	○	○	×	○
SS	×	○	×	×
CS	×	×	○	○

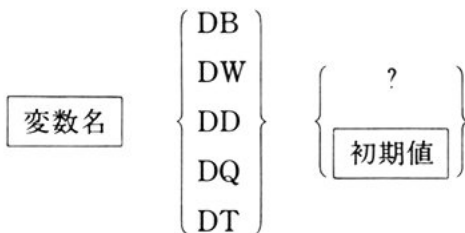
（4） 次に，Pが1かどうかを調べ，Pが0であればタイプ11の割り込みが発生する．

（5） CSの場合は，さらに新しくIPに代入されるオフセットが，新しくCSキャッシュに定義されるディスクリプタのセグメントリミットに入っているかどうかを検査する．もし，セグメントリミットを超えているときは，タイプ13の割り込みを発生する．

以上の検査にすべて合格して，初めて80286はCS，CSキャッシュおよびIPに新しい値を代入し，異なるセグメントの命令に制御を移行することができる．

このように，セグメントレジスタにはセグメントディスクリプタのタイプによって代入が許されるものと，許されないものとがある．この関係を表3･2にまとめる．

アセンブリ言語における変数定義

アセンブリ言語ASM 86およびASM 286のプログラムで変数を定義するには

の形式で定義する．DB，DW，DD，DQ，DTはメモリにそれぞれ1バイト，2バイト，4バイト，8バイト，10バイトの領域を定義することを表す．その右側に？を書いたとき，初期値を定義しないことを表し，初期値を指定したときはその値がメモリに定義される．変数名は省略してもかまわないが，オフセット10に定義した1バイトのデータをALに代入するという命令を書くとき

```
MOV AL,BYTE PTR 10
```

と書くよりは

```
BDATA DB 123
```

のように変数を定義しておき

```
MOV AL,BDATA
```

のように書く方がよい．変数名を使うことによって，データのオフセットとタイプの定義をアセンブラにまかせることができる．

3-4　メモリ参照の保護

セグメントレジスタとセグメントキャッシュに正しい値が代入された後，80286はセグメントキャッシュのディスクリプタを使用して，メモリ参照を実行することができる．ただし，DS, ESに0を代入したとき，DSキャッシュ，ESキャッシュはそれぞれ無効となり，これらを使用したメモリ参照を実行すれば，タイプ13の割り込みを発生する．SS，CSに0が代入されていることはありえない．80286はメモリ参照の正当性を検査する判断基準として，セグメントキャッシュの値を使用する．

セグメントキャッシュのリミットは，セグメントの大きさを定義する．80286のセグメントの最大値は，2^{16}(64K)バイトであるが，ディスクリプタのリミットとメモリ参照の種類によって図3·11に示す領域だけが使用可能となる．64Kバイトのセグメント内で使用できない領域には，他のセグメントを定義することもできるし，使用しないままにしておくこともできる．実行可能なコードセグメントは，オフセットが0からリミットまでが参照可能となる．したがって，表3·3に示すように，コードフェッチにおいてIPは

$$0 \leqq IP \leqq \text{リミット}$$

の値をもたなければならない．

アクセスライトのEが0で，実行できないデータセグメントの場合，EDが0のとき，オフセットが0からリミットまでが参照可能であり，EDが1のとき，オフセットが[リミット+1]から0FFFFHまでが参照可能である．一般に，DS, ESで参照するデータセグメントにはEDを0とし，SSで参照するスタックセグメントにはEDを1にする．もし，それぞれのメモリ参照において，オフセットが参照可能な範囲に入っていなければ，表3·3に示すように割り込みが発生する．

データ参照においては，表3·4に示すように使用するセグメントレジスタとアクセスライトの組み合わせによって，許されるものと許されないものがある．SSキャッシュに代入するアクセスライトは，E=0，W=1であることが，SSにセレクタを代入する段階で検査されるから，表3·4において，×印を付けた状況は起こりえない．

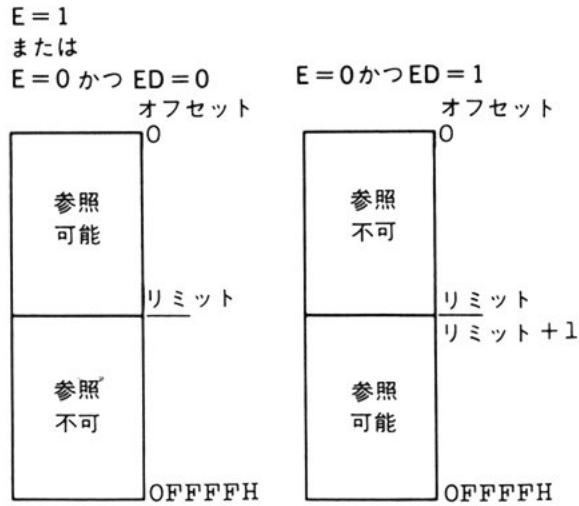



図 3・11 セグメントリミット

表 3・3 リミット検査

メモリ参照	条件	オフセット検査	例外割り込み
コードフェッチ	E＝1	0≦IP≦リミット	タイプ 13
スタックへの PUSH, POP	E＝0 W＝0 ED＝0	0≦SP≦リミット	タイプ 12
	E＝0 W＝0 ED＝1	リミット＋1≦SP≦0FFFFH	タイプ 12
データ参照	E＝0 ED＝0	0≦オフセット≦リミット	タイプ 13
	E＝0 ED＝1	リミット＋1≦オフセット≦0FFFFH	タイプ 13

表 3・4 メモリ参照におけるアクセスバイト検査

メモリ参照 ＼ アクセスライト		E＝0		E＝0	
		W＝0	W＝1	R＝0	R＝1
DS, ES を使用	データリード	○	○	例外割り込み タイプ 13	○
	データライト	例外割り込み タイプ 13	○	例外割り込み タイプ 13	例外割り込み タイプ 13
SS を使用	データリード POP	×	○	×	×
	データライト PUSH	×	○	×	×

3-5 仮想記憶

〔1〕 **実メモリと仮想メモリ**　80286は，GDT, LDTの2つのディスクリプタテーブルによってセグメントを管理する．セグメントセレクタの上位14ビットを使用して，GDT, LDTの識別と，テーブル内のディスクリプタの識別を行うから，最大2^{14}までのセグメントディスクリプタを定義することができる．各ディスクリプタに対応して，最大2^{16}(64 K)バイトの大きさのセグメントが定義できるから，80286の論理アドレスでは，**最大$2^{14}\times 2^{16}=2^{30}$(1 G)バイト**までのメモリ空間を表現することができる．

一方，80286のアドレスバスは24ビットであり，ハードウェアに実装できる最大の実メモリは**2^{24}(16 M)バイト**までである．したがって，論理アドレスで表現できるすべてのメモリ空間を実メモリに入れることはできない．

しかし，図3·12に示すように論理アドレス空間を磁気ディスク装置などの高速大容量の補助記憶装置におき，現在必要なセグメントを実メモリに入れるという方法で，OSを作ることができる．このとき，実際には80286は最大16Mバイトまでのメモリをもつことができるが，ユーザは仮想的に1Gバイトのメモリがあるようにプログラムを書くことができる．したがって，プロテクトモードでは論理アドレスのことを**仮想アドレス**とも呼ぶ．

〔2〕 **仮想記憶の実現**　当然，プログラム実行中に実メモリ上にないセグメントを参照する場合がある．このときは，実メモリ上の未使用領域を探し，もし十分な未使用領域がなければ，現在，実メモリ上にある最も必要でないようなセグメントをディスクに待避し，代わりにディスクから必要なセグメントを実メモリに入れる．このような処理を**メモリスワップ**と呼ぶ．

80286は，図3·13に示すように，アクセスライトのP, Aの2ビットを使用して，メモリスワップを実現することができる．すなわち，セグメントが実メモリ上に存在するとき，そのアクセスライトのPを1にしておく．逆に，セグメントが実メモリ上に存在しないときPを0にしておく．このように，セグメントディスクリプタを定義しておけば，80286は，P＝0であるディスクリプタをセグメントキャッシュに代入せずに，タイプ11の割り込みを発生する(ただし，SSキャッシュに代入するとき，P＝0であればタイプ12の割り込みを発生する)．

たとえば，図3·14に示すように，DSにセレクタ1BHを代入するとき，対応するディスクリプタのPが0であれば，タイプ11の割り込みを発生する．この割り込み処理において，不必要なセグメントをディスクに待避し，逆に必要なセグメントをディスクから実メモリに代入する．さらに，ディスクリプタのPに1を書いてから，IRET命令によって，割り込みを発生した命令を再実行することができる．図3·14に示すように，メモリスワップの作業は，Pの状態によって80286が自動的に発生する割り込み処理において実現されるから，ユーザはメモリスワップの存在を知らずに仮想アドレスを使用してプログラムを書くことができる．

メモリスワップにおいて，どのセグメントをディスクに待避してどのセグメン

図 3・12　仮想メモリ

Present
Pが1のとき，セグメントが実メモリに存在することを表す．逆にPが0のとき，セグメントが実メモリに存在しないことを表す．
Pを0にするか，1にするかはプログラム(OS)によって処理する．
実メモリに存在しないセグメントのディスクリプタをセグメントキャッシュに転送しようとすれば，タイプ11の例外割り込みが発生する．

Accessed
ディスクリプタがセグメントキャッシュに転送されたとき，80286は自動的にAを1にする．
Aを0にするのはプログラム(OS)で実行する．定期的にAビットの値を読むことによって，セグメントの参照頻度をカウントすることができる．

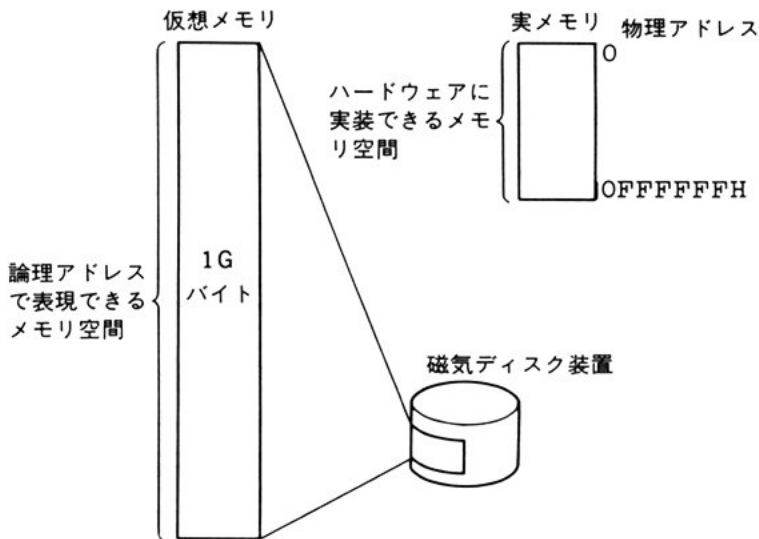
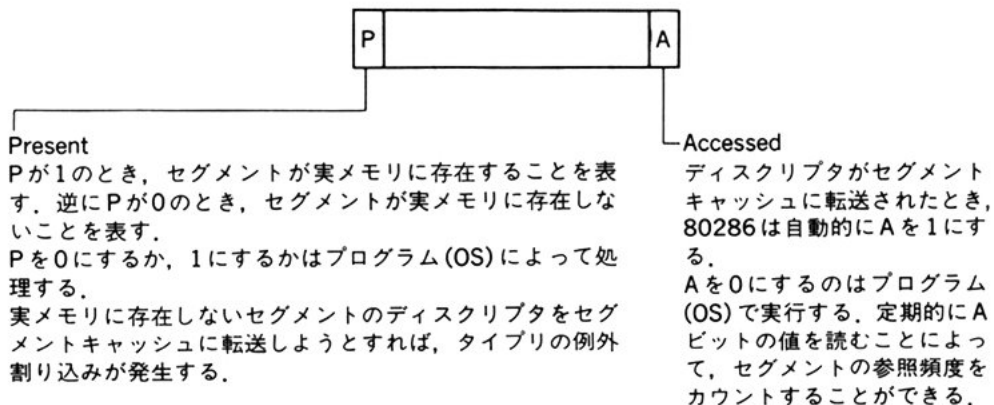



図 3・13　アクセスライトPビット，Aビットによる仮想記憶のサポート

トをメモリにおくかという管理はOSの中で処理されるが，システムのスピードに影響する重要な作業である．ひんぱんに参照されるセグメントは実メモリにおき，参照回数の少ないセグメントをディスクに待避したほうがよい．ここで，アクセスライトのAを使用して，セグメントの参照回数をカウントすることができる．Aは0に初期設定しておく．80286はディスクリプタをセグメントキャッシュに代入するとき，自動的にAに1を書く．したがって，定期的にディスクリプタのAの値を調べ，その後，Aを再び0に初期設定するような処理によって，セグメントの参照の回数を計ることができる．

仮想メモリを利用すれば，ユーザは仮想的に1Gバイトまでの非常に大きなメモリ空間を利用することができる．しかし，仮想記憶はディスクと実メモリとの間でのメモリスワップを伴うため，システムの処理速度の低下がある．したがって，仮想記憶は複数のユーザが時分割でシステムを利用するマルチユーザシステムには有効であるが，処理のタイミングが重要なリアルタイムシステムには使用すべきではない．80286において，仮想記憶を使用しないときは，ディスクリプタのPを常に1にしておけばよい．

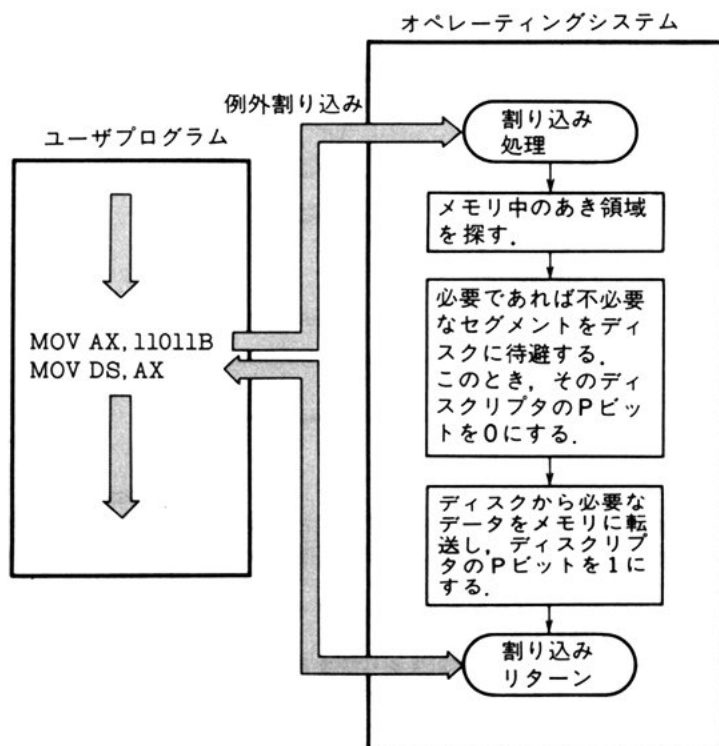



図 3・14　Pビットによる例外処理

3-6 ディスクリプタテーブルの扱い

OSの処理では，ディスクリプタテーブルに新しいディスクリプタを追加したり，必要でないディスクリプタを無効にしたり，また，**3-5**で述べたようにアクセスライトを必要に応じて書き換えることがある．ディスクリプタテーブルを変更する方法は簡単である．ディスクリプタテーブルを1つのデータセグメントとして扱えばよい．たとえば，図3·15に示すようにGDTのスロット1に，GDTをデータセグメントとして扱うためのセグメントディスクリプタをあらかじめ定義しておく．このディスクリプタのアクセスライトをE＝0，W＝1のように設定しておけば

```
MOV AX,1000B
MOV ES,AX
```

のように，ESにセレクタ8を代入することによって，ESを使用してGDT内のデータを変更することができる．たとえば，図3·15のようにベースアドレス0から始まり，48バイトの大きさをもつGDTがすでに定義されているとすると，

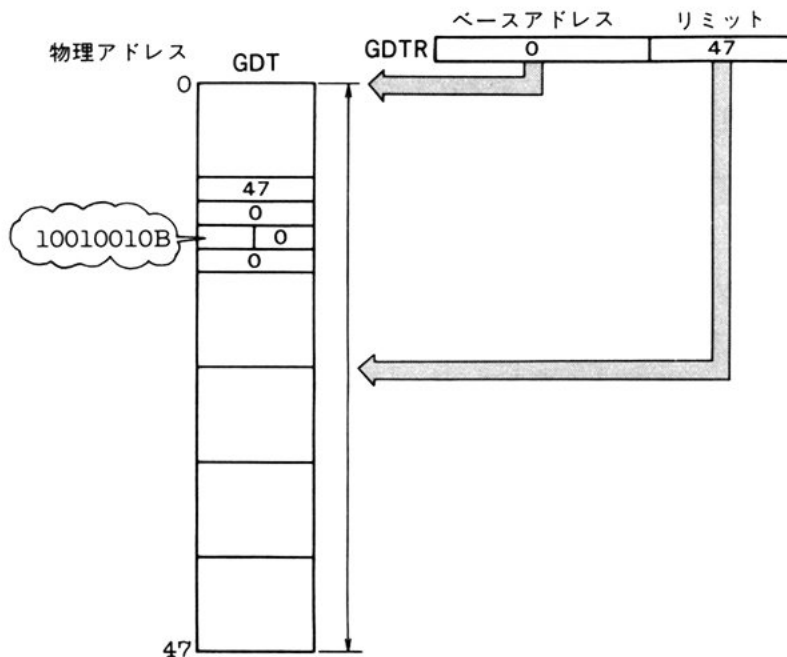



図 3·15 予備ディスクリプタ（alias）

MOV AX,1000B MOV ES,AX	➡GDT予備ディスクリプタのセレクタ＝8HをESに代入する.
ADD ES:WORD PTR 8,8	➡ES：8の1ワードのメモリに8を加算する. GDT予備ディスクリプタのリミットは47から55になる.
MOV AX,1000B MOV ES,AX	➡ESキャッシュのリミットも55にするため再びESにGDT予備ディスクリプタのセレクタを代入する.
MOV BX,48	➡BXに新しいディスクリプタのオフセットを代入する.
MOV WORD PTR ES:[BX]+0,LIMIT	➡新しいディスクリプタのリミットフィールドに新しいセグメントのリミットを定義する.
MOV WORD PTR ES:[BX]+2,BASEADDRE_LOWER	➡新しいディスクリプタのベースアドレスフィールドの下位ワードに新しいセグメントのベースアドレスの下位ワードを定義する.
MOV WORD PTR ES:[BX]+4,BASEADDRE_UPPER	➡新しいディスクリプタのベースアドレスフィールドの上位バイトに新しいセグメントのベースアドレスの上位バイトを定義する.
MOV WORD PTR ES:[BX]+5,AR_BYTE	➡新しいディスクリプタのアクセスバイトに新しいセグメントのアクセスバイト値を定義する.
MOV WORD PTR ES:[BX]+6,0	➡新しいディスクリプタの最上位ワードに80386との互換性のために0を代入する.
LGDT QWORD PTR ES:[BX]	➡GDTRに新しいGDTのベースアドレスとリミットを代入する.

図3・16 GDTの変更処理

GDTのオフセット48から始まる新しいディスクリプタを追加するプログラムは図3·16のようになる.

一般には，OSを設計する場合，ディスクリプタを必要なたびに新しく作るのではなく，前もって見積もっただけの大きさのディスクリプタテーブルを定義したほうがよい．このとき，使用しないディスクリプタのアクセスライトには0を書いておく．アクセスライトが0のとき，そのディスクリプタをヌルディスクリプタと呼び，もし間違ってヌルディスクリプタをセグメントキャッシュに代入しようとすれば，80286は自動的にタイプ13の割り込みを発生するので，システムの誤まりを発見することができる.

3-7 プロテクトモードの初期設定

以上見たように，80286をプロテクトモードで動作させるためには，メモリに少なくともGDTを定義し，GDTのベースアドレス，リミットをGDTRに代入しなければならない．これらの処理は80286がリアルモードで動作中に実行する必要がある．GDTを定義する方法は2つある．

1つの方法は，あらかじめROMに書き込んだGDTをRAMにコピーする．または，ディスクファイルに定義したGDTの初期データをメモリに代入する方法がある．

ここでは，図3·17に示すように物理アドレスOFF000Hから4Kバイトの大きさのGDTの初期データがROMに書き込まれているとして，物理アドレス

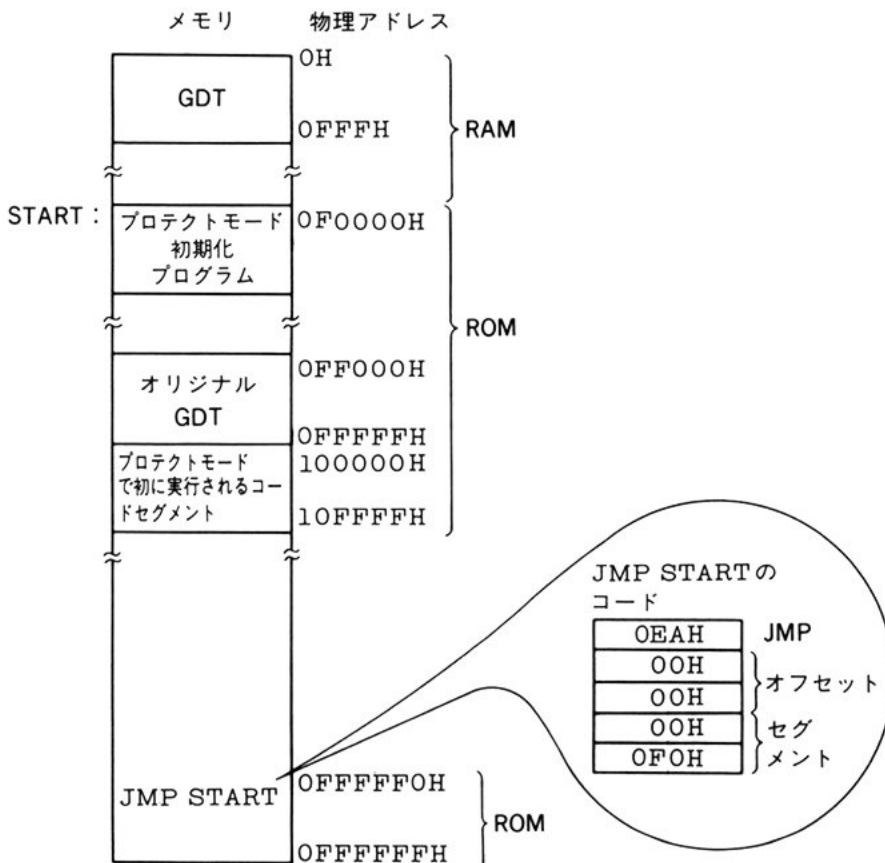



図 3·17　GDTの初期設定

0H から始まる 4K バイトの大きさのメモリに GDT を定義し，プロテクトモードに切り換えてから，物理アドレス 10000H から始まるプログラムを実行するまでの処理の流れについて考える．ここで，物理アドレス 0FF000H から始まる ROM には，図 3･18 に示すような GDT の初期データが定義されているものとする．図において，オフセット 8 から始まる 8 バイトの領域に GDT の補助ディスクリプタが，オフセット 24 から始まる 8 バイトの領域にプロテクトモードで最初に実行されるコードセグメントのディスクリプタが定義されている．

80286 をリセットしたとき，リアルモードで物理アドレス 0FFFFF0H にある命令を最初に実行する．ここには一般に図 3･17 に示したように，最初に実行

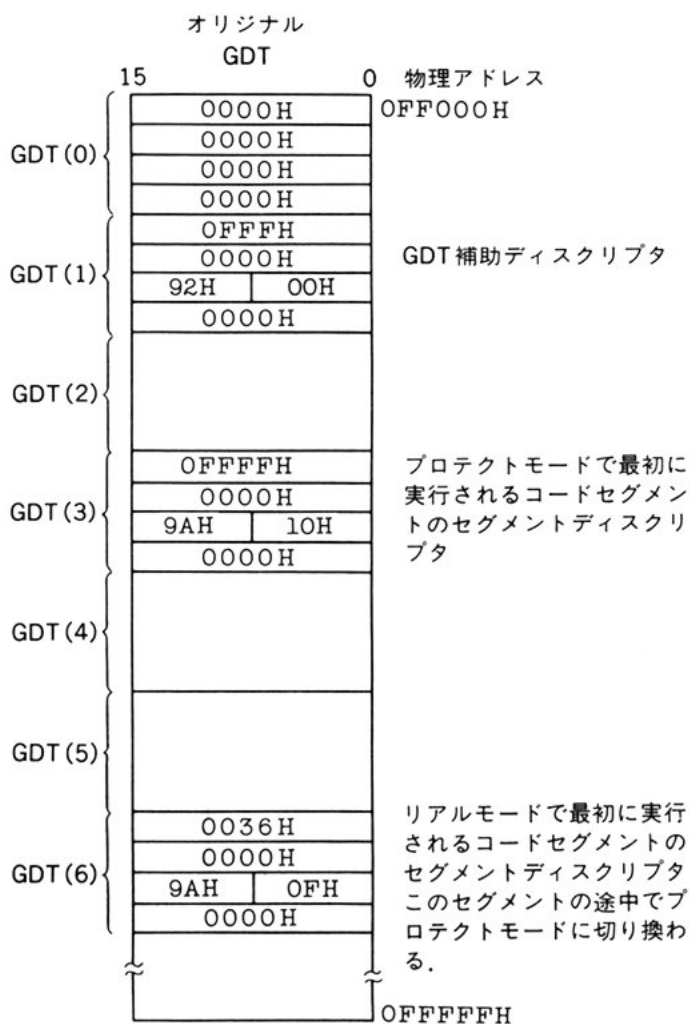



図 3・18　ROM 化してオリジナル GDT

されるプログラムに制御を移すJMP命令をおく．このJMP命令で，セグメントセレクタがOF000H，オフセットが0のアドレスへジャンプし，図3・19に示すプロテクトモード初期化プログラムを実行する．このとき，アドレスバスとA_{20}-A_{23}はOFHから0に変わることに注意する．

ここでは，GDTの初期化だけについて考えたが，プロテクトモードにスイッチする前にIDTの初期化も行う必要がある．IDTについては第6章において述べる．また，LDTの初期化はプロテクトモードになってから実行する．なぜなら，LGDT命令はリアルモードでも，プロテクトモードでも使用できるのに対して，LLDT命令はプロテクトモードでしか使用できないからである．

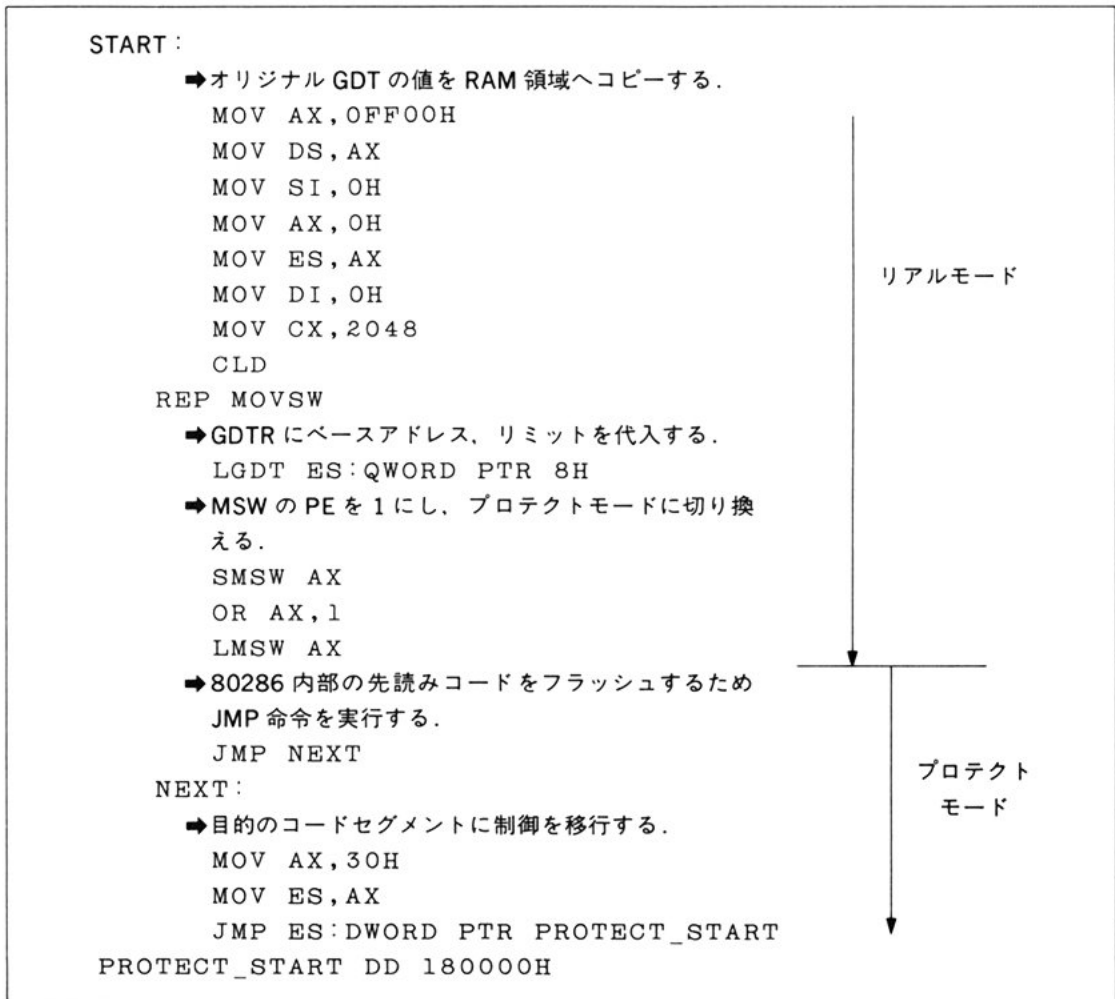
```
START:
      ➡オリジナルGDTの値をRAM領域へコピーする．
        MOV AX,0FF00H
        MOV DS,AX
        MOV SI,0H
        MOV AX,0H
        MOV ES,AX
        MOV DI,0H
        MOV CX,2048
        CLD
    REP MOVSW
      ➡GDTRにベースアドレス，リミットを代入する．
        LGDT ES:QWORD PTR 8H
      ➡MSWのPEを1にし，プロテクトモードに切り換
        える．
        SMSW AX
        OR AX,1
        LMSW AX
      ➡80286内部の先読みコードをフラッシュするため
        JMP命令を実行する．
        JMP NEXT
    NEXT:
      ➡目的のコードセグメントに制御を移行する．
        MOV AX,30H
        MOV ES,AX
        JMP ES:DWORD PTR PROTECT_START
  PROTECT_START DD 180000H
```


図 3・19 プロテクトモード初期化プログラム

4. 特権レベルによる保護

マイクロコンピュータシステムのソフトウェアは，大きく分類して **OS**（オペレーティングシステム）または**スーパバイザ**と呼ばれる管理プログラムと，その下で働くさまざまな**応用プログラム**（**ユーザプログラム**）によって構成されている．このようなシステムにおいて，1つのユーザプログラムが暴走しても，OSの処理に影響しないようにするためにはどうすればよいだろうか．また，第3章で見たようにGDT, LDTのディスクリプタを書き換えることは，システムの動作に致命的な影響を及ぼす．ユーザプログラムがGDT, LDTを勝手に書き換えないようにするためにはどうすればよいだろうか．80286では各セグメントに特権レベルを付けて階層化し，一部の不完全なプログラムによってOSなどのシステムに重要な処理またはデータが壊されるのを防ぐことができる．

4-1 特権レベル

〔1〕 **OSの保護**　OSは，ファイル管理，メモリ管理，タスク管理などのプログラムによって構成される．ユーザプログラムからは必要に応じて，OSのもつこれらの機能をサブルーチンコール（手続きの引用）という形で利用することができる．簡単のために，図4·1に示すようなシステムを考える．OSの中にFREADという手続きが定義されている．FREADは，ユーザプログラムからファイルの識別データ，メモリのポインタなどをパラメータとして受け取り，ファイルから入力したデータをユーザプログラムのバッファ領域に転送する処理である．

図4·1に示す例から，OSとユーザプログラムの間には一般に次のような関係が成り立つ．

（1） ユーザプログラムから，OS内部のサブルーチンへパラメータを渡してコールできなければならない．しかし，ユーザプログラムの暴走によって誤ってOS内部に制御が移らないように，制御の移行を検査できる機能が必要である．また，サブルーチンリターン以外に，OSからユーザプログラムに制御を移行する必要はない．なぜならば，OSレベルのプログラムが，OS内部のプログラムより信頼性の低いと思われるユーザ定義の手続きを引用することはないからである．

（2） OS内部のプログラムからユーザプログラム内部のデータセグメントは自由に参照できなければならない．逆に，ユーザプログラムがOS内部のデータセグメントを参照することは禁止するべきである．

このように，ユーザプログラムに対して，OSだけに特権を与えることによって信頼性の高いシステムを作ることができる．特権は**特権レベル**と呼ばれる番号によって区別する．OSを設計する場合，図4·1の例に示したように，特権レベルは2レベルあればなんとか間に合う．UNIXまたはXENIXがカーネルとユーザプログラムの2レベルの特権を使用している例である．しかし，OS内部においても，カーネルまたはニュークリアスと呼ばれるOSの最も中心となる部分と，デバイスドライバなどの部分を特権によって分離したい場合がある．さらに，データベースシステムのような本来のOSには属さないが，ユーザプログラムよ

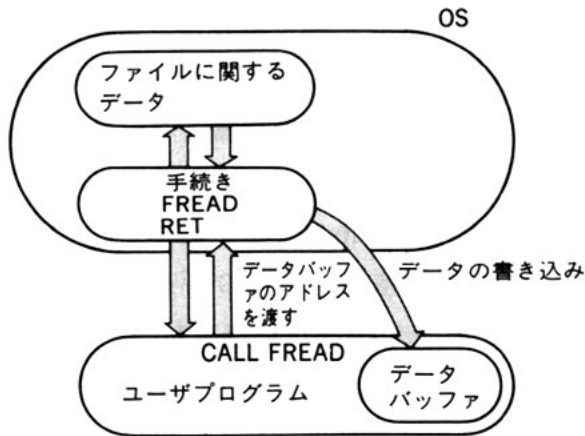



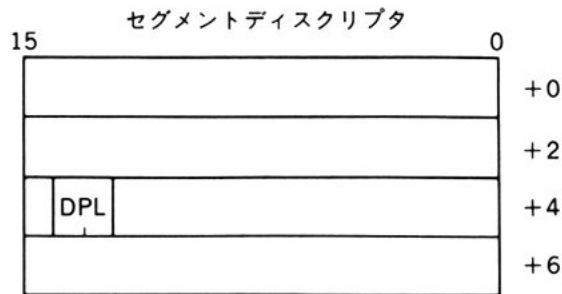
図 4・1　ユーザプログラムと OS

セグメントの特権レベルはセグメントディスクリプタのアクセスライト中のDPLによって定義される.

図 4・2　セグメントの特権レベルの定義

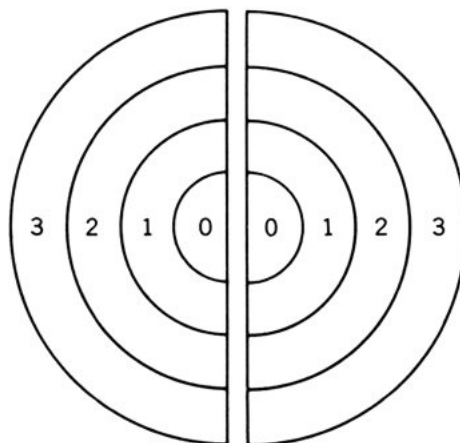



図 4・3　特権レベルによるセグメント管理

りも上の特権に設定したいプログラム群もある．

［2］ **80286 の特権保護**　80286 では，上述のようなあらゆる種類の OS 設計に柔軟に対処できるように，4 レベルの特権をセグメントごとに定義することができる．セグメントの特権レベルは，図 4·2 に示すようにセグメントディスクリプタの **DPL** (descriptor privilege level) の 2 ビットに，0 から 3 までの数値で定義する．数値的に最も小さい特権レベル 0 が最も大きな特権をもち，数値的に最も大きな特権レベル 3 が最も小さい特権をもつ．特権レベルを考慮すれば，80286 の論理アドレス空間は図 4·3 に示すように表現することができる．

特権レベルの利用は，図 4·4 に示すように OS の最も中心となるカーネルを特権レベル 0 に定義する．これは，LGDT 命令，LLDT 命令など，80286 の動作に根本的に影響する命令は特権命令と呼ばれ，特権レベル 0 でしか使用できない

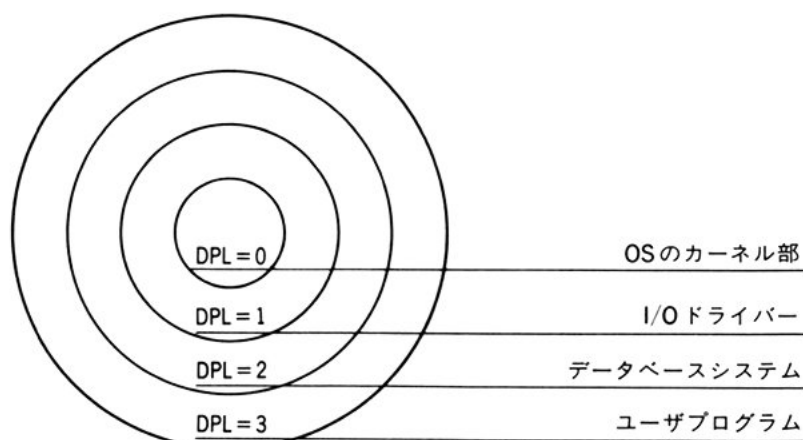



図 4・4　80286 で定義できる特権レベル

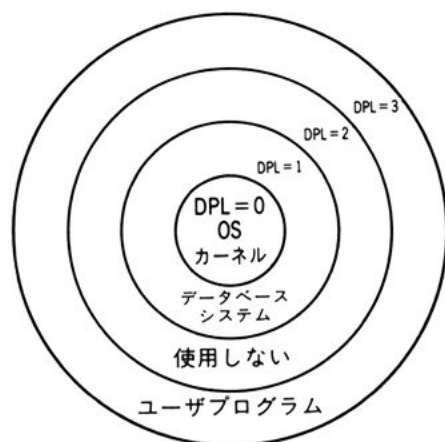



図 4・5　OS 設計に必要な特権レベル

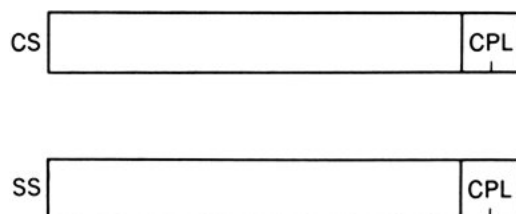



図 4・6　CPL (現在の特権レベル)

からである．その他のプログラムは特権レベル0以外の特権を任意に利用してよいが，一般にユーザプログラムは特権レベル3に定義する．図4·4に示すように，特権レベル1にI/Oドライバ，特権レベル2にデータベースシステムというように定義してもよい．しかし，必要もなくすべての特権レベルを使用するべきではない．図4·5に示すように，デバイスドライバをカーネルと同じ特権レベル0に定義し，データベースシステムを特権レベル1に，ユーザプログラムを特権レベル3に，それぞれ定義してもよい．XENIX 286は特権レベル0と3を使用し，他の特権レベルは使用していない．また，特権による保護が必要なければ，すべてのセグメントを特権レベル0に定義すればよい．

80286は図4·6に示すように，CS，SSの下位2ビットとディスクリプタのDPLを比較することによって，特権による保護を実行する．CS，SSの下位2ビットは常に同じ値であり，**CPL**（現在の特権レベル）と呼ぶ．80286が特権による保護を実行するとき，CPLの値を特権保護の評価基準として使用する．

メモリ管理の方法

メモリ管理とは，プログラムが使用してよい領域といけない領域を識別できる機能である．このために，セグメンテーションとページングの2つの方法がある．

セグメンテーションは，コード領域，データ領域，スタック領域などのセグメントと呼ばれる論理的単位でメモリの部分集合を扱う．ディスクリプタテーブルと呼ばれる特別なテーブルによって各セグメントのアドレス，大きさなどが管理される．ページングは，記録されるデータの論理的意味に関係なく，ページと呼ばれる一定の大きさ（たとえば4Kバイト）の単位にメモリを分割して管理する．

セグメントはその大きさを柔軟に定義できるが，ページは固定長であるから，ページングによってメモリの論理的管理までもたせることは不利である．しかし，仮想メモリのシステムにおいてメモリとディスクとの間でデータの交換を行う処理には固定長のページは適している．

8086のメモリはセグメンテーションを採用するが，ディスクリプタテーブルによる管理機能のない限定されたものである．80286はディスクリプタテーブルによって管理されるセグメンテーションをもつ．そして，80386は80286と同様のセグメンテーションに加えてページング機能をもつ．

4-2　データセグメント，スタックセグメントの特権保護

図4·7に示すように，特権レベルnのデータセグメントD_SEGに定義された変数VARに定数3を代入する処理を考える．3つのMOV命令が，特権レベルnのコードセグメントC_SEG，nより数値的に小さい特権レベル1のコードセグメントA_SEG，そして，nより数値的に大きい特権レベルのコードセグメントB_SEGにそれぞれ定義されている．この中で，B_SEGから変数VARを参照することは許されない．これは，**4-1**で述べた「ユーザプログラムがOS内のデータセグメントを参照することは禁止するべきである」という保護を，80286が特権レベルの値に応じて実行するからである．

たとえば

```
MOV  DS,AX
```

図 4·7　データセグメントの特権保護

の命令で，DS にセグメントセレクタを代入するときの状況を図 4･8 に示す．このとき，**3-3** で述べた保護に加えて

DPL ≧ MAX(CPL,RPL)

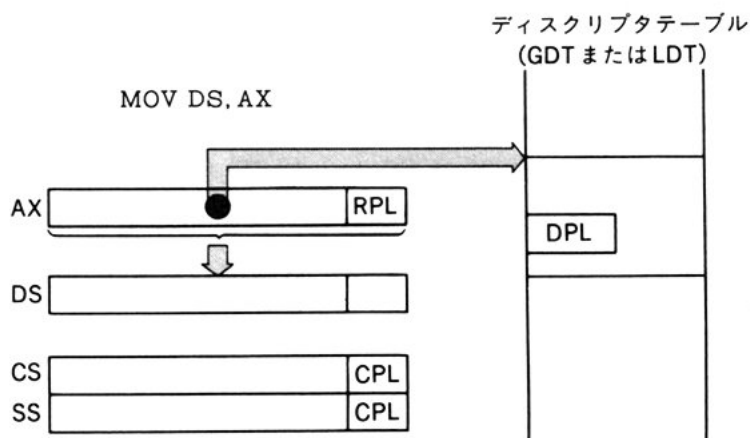
の関係を満足しなければならない．ただし，MAX(　)は括弧の中のパラメータに指定した最も大きな数値を表すものとする．ここで，CPL は MOV　DS，AX を実行する現在の特権レベルを表し，RPL は新しいセレクタを DS に代入することを要求するコードセグメント(すなわち，MOV　AX，セグメントセレクタを実行したコードセグメント)の特権レベルを表す．多くの場合，CPL＝RPL と考えられるが，CPL≒RPL の場合もある．このとき，80286 自身が実行する

特権について，DPL≧CPL かつ DPL≧RPL
の条件を満たさなければならない．

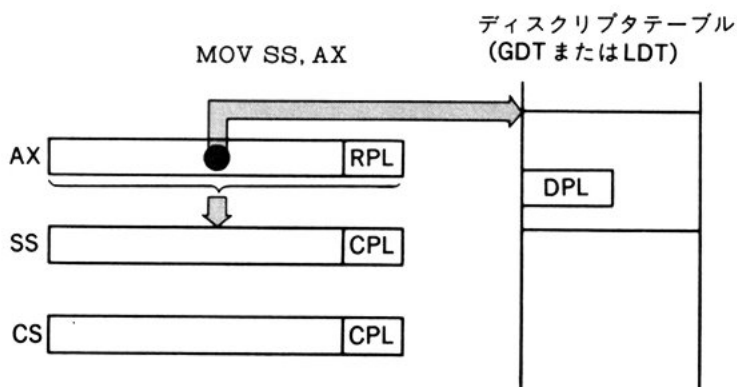
図 4・8　DS または ES の特権保護

特権について，DPL＝RPL かつ DPL＝CPL
の条件を満たさなければならない．

図 4・9　SS の特権保護

特権保護だけでは完全ではない．しかし，今はRPL＝CPLと考えてよい．RPLとCPLが異なる場合については**4-8**で述べる．上述の条件は，実行中のコードセグメントの特権レベルより大きな特権の（特権レベルが数値的により小さい）セグメントで定義されたデータを参照できないことを表す．もし，この条件が満足されないときは特権保護によって80286は，タイプ13の割り込みを発生する．このときDSおよびDSキャッシュの値は変化しない．

以上のことはESについてもDSの場合とまったく同様である．

次にスタックセグメントの保護について考える．たとえば，図4·9に示すように

```
MOV  SS,AX
```

の命令によって，新しいセグメントセレクタをSSに代入する．このとき

```
DPL = CPL = RPL
```

の関係が成立しなければ，80286はSSおよびSSキャッシュの値を変更せずにタイプ13の割り込みを発生する．すなわち，スタックセグメントの特権レベルは，常に現在実行中のコードセグメントの特権レベルに等しくなければならない．

4-3 コードセグメントの特権保護

CSを書き換える命令には，リアルモードの場合と同じようにfar JMP，far CALL，そして割り込みがある．割り込みについては第6章で述べるので，ここではfar JMPとfar CALLについて考える．図4・10にfar CALLを使用してコードセグメントCODE 2から，コードセグメントCODE 1に定義された手続きPROC 1に制御を移行する例を示す．CALL PROC1の命令を実行するときの状況を図4・11に示す．far CALLは，5バイトの命令で，制御を移行する先のセグメントセレクタとオフセットをそれぞれ2バイトで表す．オフセットがIPに，またセグメントセレクタがCSに代入されるが，このとき

RPL ≦ CPL かつ CPL = DPL

の条件を満足しなければならない．もし，上述の条件を満足しなければ80286はCALL命令を実行せずにタイプ13の割り込みを発生する．すなわち，一般のfar CALLでは，特権レベルの異なるコードセグメントへ制御を移行することは

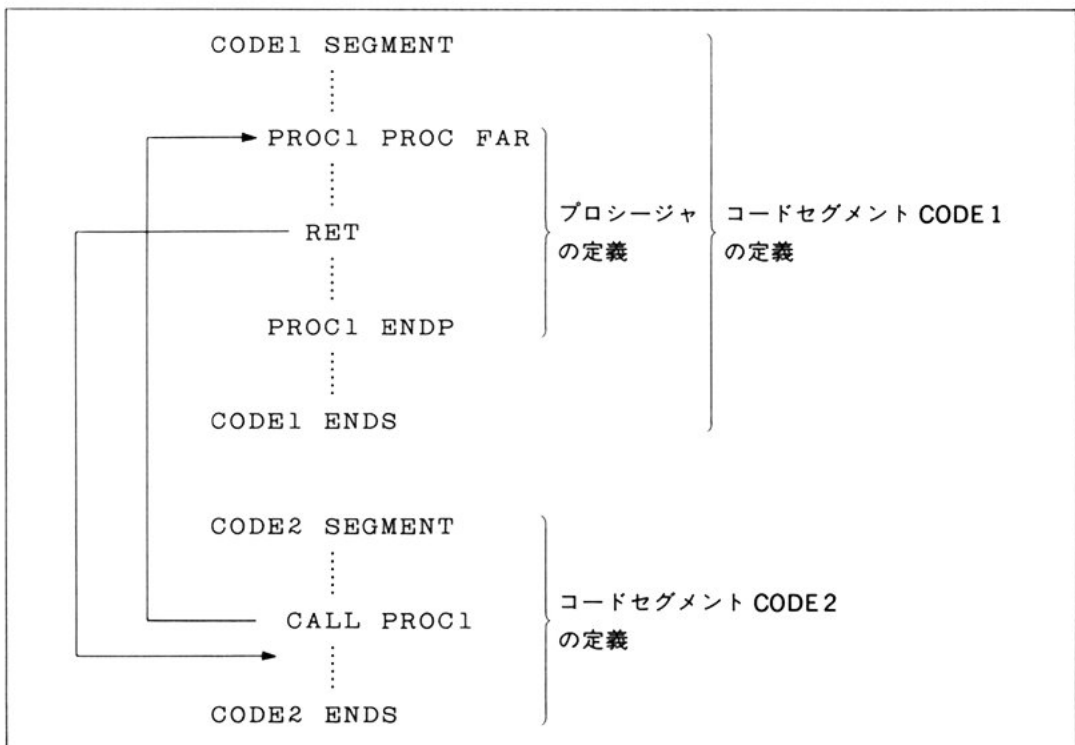



図 4・10 セグメント外への制御移行

できない．したがって，特権の低いプログラムが暴走しても，特権レベルの異なるプログラムに誤って制御が移行することがない．このことはCALL命令が制御を移行する前に戻りアドレスをスタックにPUSHすることを除けば，far JMP命令についてもまったく同じである．

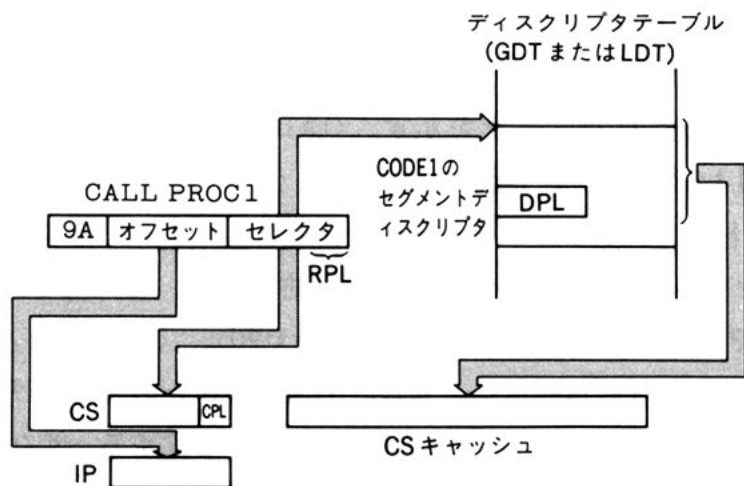



図 4・11　CSの特権保護

4-4 コールゲートによる制御移行

CPLを変更して，より高い特権レベルのコードセグメントに制御を移行する場合，**コールゲート**と呼ばれる特殊なディスクリプタを使用する．コールゲートは，セグメントディスクリプタのようにセグメントの属性を定義するものではなく，より大きな特権のコードセグメントで定義された手続きへの制御移行の入口を定義するディスクリプタである．コールゲートは，図4・12の形式をもち，GDTまたはLDTに定義する．プログラムでコールゲートを参照する場合，そのセレクタによって識別することができ，far CALL命令またはfar JMP命令のセレクタ部（5バイトの命令コードの最上位の2バイト）にコールゲートのセレクタを定義することができる．このとき，far CALL命令およびfar JMP命令は，制御を移行する前にコールゲートを読み，コールゲートに定義されたデータに従って動作する．

図4・12に示すように，コールゲートには制御を移行するセグメントセレクタとオフセットを下位2ワードに定義するが，セグメントセレクタのRPLの2ビットは使用しない．**ワードカウント**には0から31の値を指定することができ，ここにはスタックを介して手続きに渡すパラメータのワード数を定義する．コールゲートのP，DPLは，データセグメントに対するものと同じ扱いである．なぜならば，コールゲート自体がCALL命令，JMP命令で参照される特殊なデータと考えられるからである．すなわち，コールゲートを参照するfar CALL命令，far JMP命令は，CPL ≦ DPLとなるようなCPLのコードセグメントで実行されなければならず，またPは1でなければならない．

図4・13に，コールゲートを参照するCALL命令と，それによってCS，IPの値が変わる様子を示す．図において，EXTRN CGATE:FARは，コールゲートCGATEが他のファイルで定義され，CGATEを参照する命令をfarタイプにすることを表す．もし，CGATEがGDT(5)で定義されているとすると，CALL CGATEのオブジェクトコード（機械語）は，9AH,00H,00H,08H,02Hの順にアドレスの小さいほうからメモリに配置される．

このCALL命令を実行したとき，GDT(5)から読んだディスクリプタがコールゲートであることを80286が発見すれば，コールゲートに定義されたオフセッ

トをIPに代入し，またコールゲートに定義されたセグメントセレクタをCSに代入し，さらにCSで指定されるセグメントディスクリプタをCSキャッシュに代入する．このとき，制御を移行するセグメントディスクリプタのDPLに関して

CPL ≧ DPL

が成立するならば，DPLが新しいCPLになる．なお，far CALLのセレクタが

Xは80286がその値を無視することを表す．

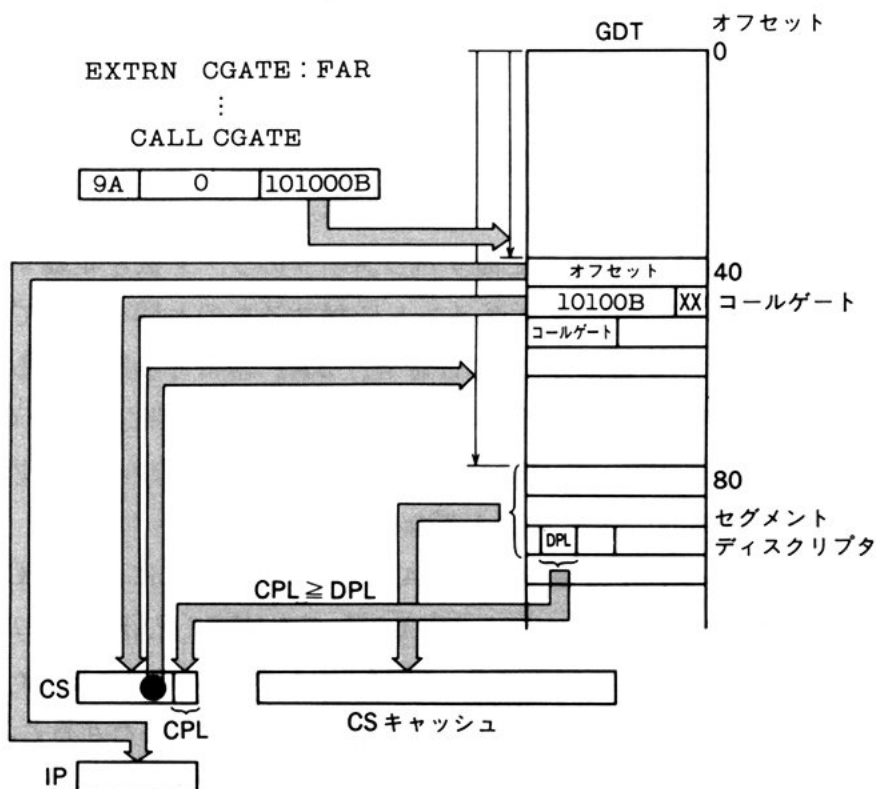
図 4・12 コールゲート

注） EXTRNは，その右に指定したシンボルが他のファイルで定義されタイプがFARタイプであることを表すアセンブリ言語の疑似命令である．このとき，CALL CGATEはfar CALL命令となる．

図 4・13 コールゲートを介した制御移行

コールゲートを指定する場合は，命令コードの中のオフセットの値は使用されない．このように，コールゲートを参照するCALL命令によって，CPLを変更してより高い特権レベルのセグメントへ移行することができる．

JMP命令のオペランドにもコールゲートを指定することはできるが，JMP命令の場合は，コールゲートを使用してもなお制御を移行する新しいセグメントディスクリプタのDPLに関して

CPL = DPL

の条件が必要である．したがって，JMP命令で特権レベルを変えることは不可能で，コールゲートを使用する意味はない．コールゲートを参照するCALL命令において，特権レベルが変化しなくてもかまわないが，そのような制御移行にわざわざコールゲートを使用するのはあまり意味がない．

ASM286（アセンブラ）のコールゲートの扱い

80286の命令で，オペランドにコールゲートを指定する特別なCALL命令，JMP命令があるわけではない．80286のCALL命令，JMP命令は8086のものとなんら変わるところはない．CALL命令，JMP命令の参照するセレクタがゲートを指していればゲートを介した制御移行となるし，セグメントを直接に指していればゲートを介さない制御移行となる．ASM286でプログラムするとき，CALL命令のオペランドが外部で定義されているならば

```
EXTRN OS_ROUTINE:FAR
```

のようにプログラムの先頭で宣言して

```
CALL OS_ROUTINE
```

のように書けばよい．このとき，OS_ROUTINEがコードセグメントのディスクリプタを指しているのか，コールゲートを指しているのかということを意識する必要はない．

4-5 スタックセグメントの保護

コールゲートを参照するCALL命令で特権レベルを変更したとき，80286は自動的にSSとSPの値を変更する．なぜならば，特権を切り換えても高い特権レベルのコードセグメントが低い特権レベルのコードセグメントと共通のスタックセグメントを使用していたとすれば，低い特権レベルからの影響がスタックを介して高い特権レベルのプログラムに及ぶからである．したがって，図4·14に示すように0から3までのすべての特権レベルを使用するならば，それぞれの特権レベルにそれぞれ独立なスタックセグメントを定義する．なお，コールゲートを使用したCALL命令でも，特権レベルが変わらなければスタックセグメントも変わらない．

たとえば

```
PUSH PARA1
PUSH PARA2
PUSH PARA3
CALL CGATE
```

のように，3ワード(6バイト)のパラメータをスタックにPUSHしてから，図4·15に示すようなコールゲートCGATEを介して制御を特権レベル3から特権レベル0に移行する場合について考える．図4·16にスタックセグメントの変更の様子を示す．特権レベルが変わるとき，新しい特権レベルのスタックセグメントのSS, SPの初期値を**TSS**(task status segment)と呼ばれる特別なセグメントから読み，SS, SPに代入する．この後，前のスタックセグメントのSS, SPの値を新しいスタックにPUSHし，前のスタックから新しいスタックにパラメータをコピーする．このとき，コピーするパラメータのワード数

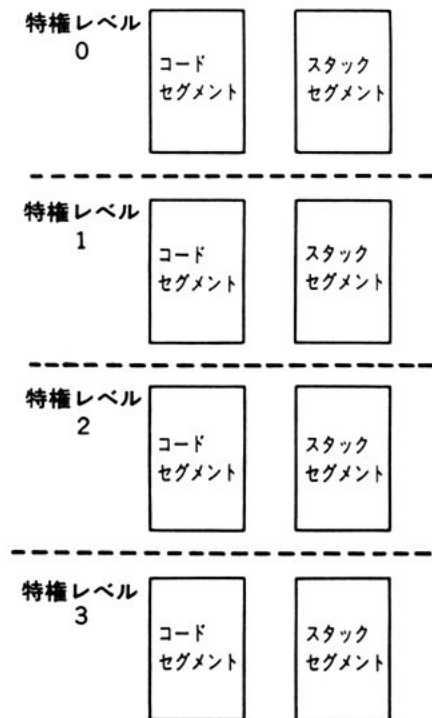



図 4·14　コードセグメントとスタックセグメントの対応

は，コールゲートのワードカウントで定義される値である．最後に新しいスタックに戻りアドレスのCSとIPをPUSHする．このように特権レベルが変わるとき，CALL命令のスタック操作は少し複雑であるが図4･16からもわかるように，新しいスタックの状態は8086のプログラムでパラメータをスタックで渡す場合と同じである．

なお，TSSのオフセット2から12までに，0から2までの各特権レベルで使用するSS，SPの初期値が定義されている．コールゲートを使用して，他の特権レベルから特権レベル3に移行することはないから，特権レベル3のSS，SPの初期値の定義は必要ない．TSSについては第6章においてより詳しく説明する．

図 4・15 CGATEの定義

```
PUSH   PAPA1
PUSH   PARA2
PUSH   PARA3
CALL   CGATE
```

特権レベル3 スタックセグメント　オフセット 0

特権レベル0 スタックセグメント　オフセット 0

TSS　オフセット

TSS	オフセット
特権レベル0 SP	+2
特権レベル0 SS	+4
特権レベル1 SP	+6
特権レベル1 SS	+8
特権レベル2 SP	+10
特権レベル2 SS	+12

CALL前SS　CALL前SP　PARA3　PARA2　PARA1

戻り IP　戻り CS　PARA3　PARA2　PARA1　CALL前SP　CALL前SS

CALL直後SS　SS

CALL直後SP　SP

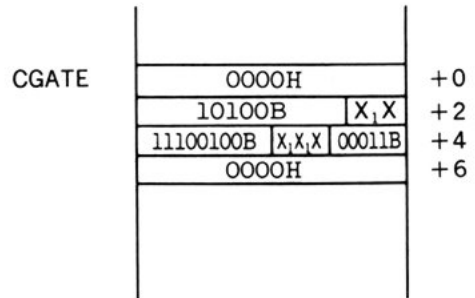
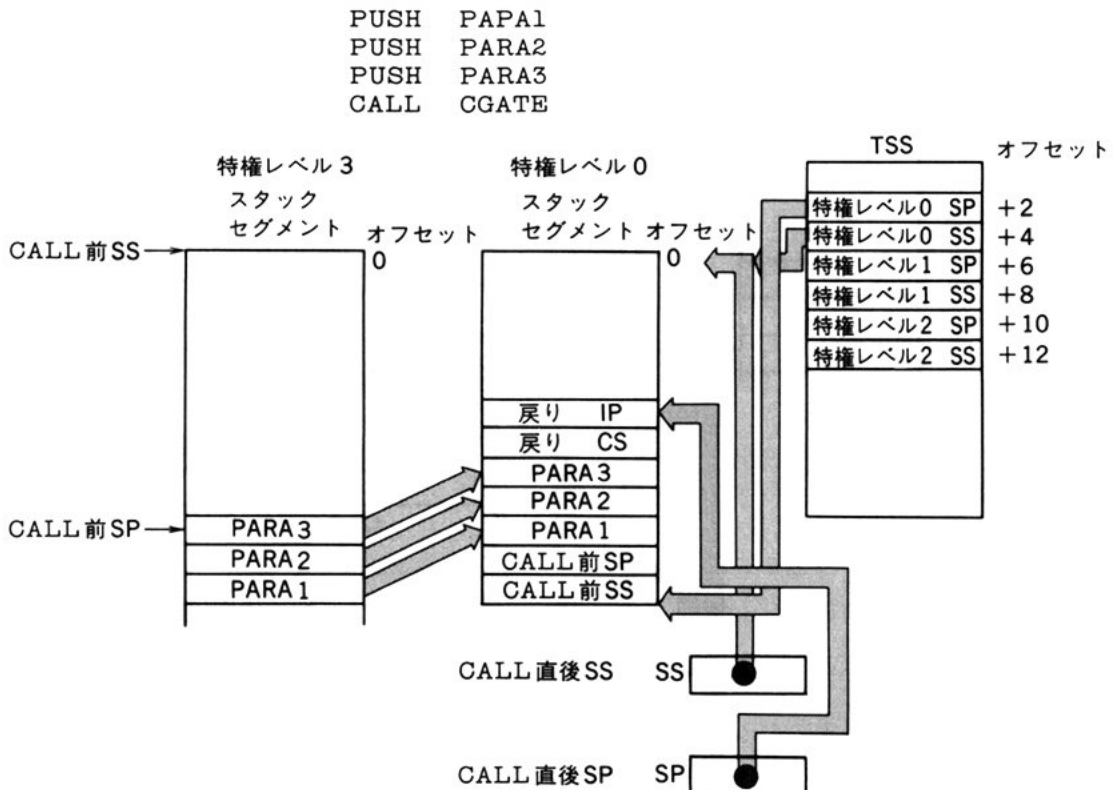



図 4・16 コールゲートによるスタックセグメントの切り換え

4-6 RET命令による制御移行

4-5で述べたように，コールゲートを使用したCALL命令によって特権レベルを変更して，制御を移行した場合にもRET命令によってCALL命令の次の命令に制御を戻すことができる．このときの様子を図4·17に示す．スタックに6バイトのパラメータがPUSHされている場合のRET命令は

RET 6

のように，オペランドにパラメータのバイト数を指定しなければならない．

まず，スタックにPUSHされている戻りCSのRPLとCPLが比較され，

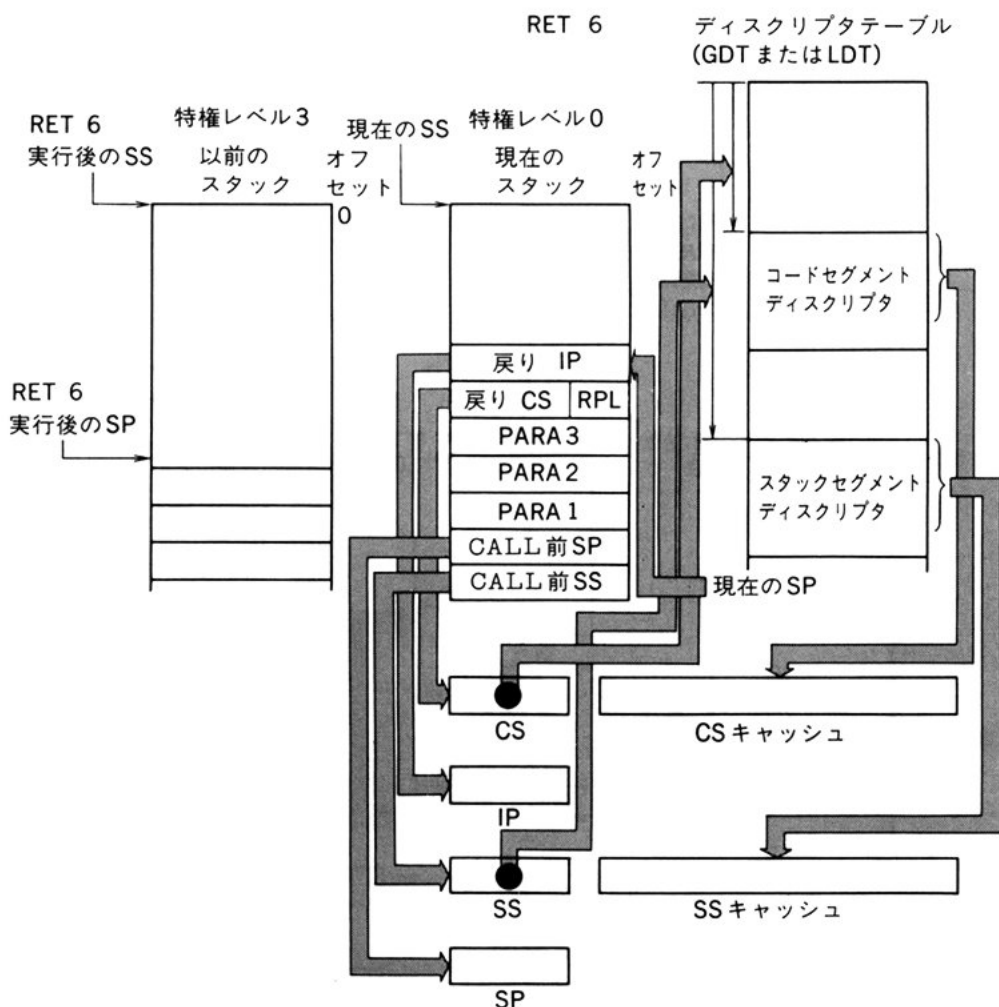



図 4·17　RET命令による制御移行

RPL と CPL が等しい場合はリアルモードのときと同じ RET 命令が実行される．また RPL が CPL より数値的に大きな場合は，CALL 命令のときと逆に特権の低いコードセグメントへ戻る．このとき，SS, SP の値も自動的に[現在の SP+4+RET 命令のオペランド]のオフセットから保存されている，CALL 前の SS, SP の値によって書き換えられる．また，CS, SS が変更されたとき，CS キャッシュ，SS キャッシュも変更され，ディスクリプタの検査も自動的に実行される．

RET 命令によって低い特権レベルに制御が戻るとき，DS, DS キャッシュ，ES, ES キャッシュの値についても検査される．もし，RET 命令で戻るコードセグメントの CPL に対して，DS または ES のキャッシュの DPL が数値的に小さい場合は，80286 は自動的に DS または ES にヌルセレクタを代入し，DS, ES を使用できないようにする．

8086, 80286 の RET 命令

8086, 80286 の RET 命令には 2 章で述べたように far RET 命令と near RET 命令の 2 種類がある．

注意すべきことは，far CALL と far RET, near CALL と near RET は必ず対応させて使用しなければならないし，また同じ手続き中の RET は far タイプか near タイプのどちらかに統一しなければならない．

しかし，RET 命令は命令のニーモニックで far と near を区別することはできない．なぜなら，far RET 命令と near RET 命令の使い分けをプログラマにまかせることは，プログラマの負担が大きくなるからである．かわりにアセンブリ言語 ASM 86, ASM 286 では

```
PROC  { FAR  }
      { NEAR }
  :
  :
ENDP
```

のように，手続きを定義する疑似命令をもつ．PROC NEAR(このとき，NEAR は省略可能)と ENDP の間に書かれた RET 命令はすべてアセンブラによって near RET と解釈され，PROC FAR と ENDP の間の RET 命令はすべて far RET と解釈される．

4-7 コンフォーミングコードセグメント

高い特権レベルのコードセグメント内部で定義された手続き，データを，必要に応じて低い特権レベルから参照したい場合がある．図 4·18 に示すように，実行可能なコードセグメントのアクセスライトの C を 1 にしたものは，コンフォーミングコードセグメントと呼び，例外的な特権保護が実行される．

コンフォーミングコードセグメントで定義された手続きに，CALL 命令，JMP 命令を用いて制御を移行する場合

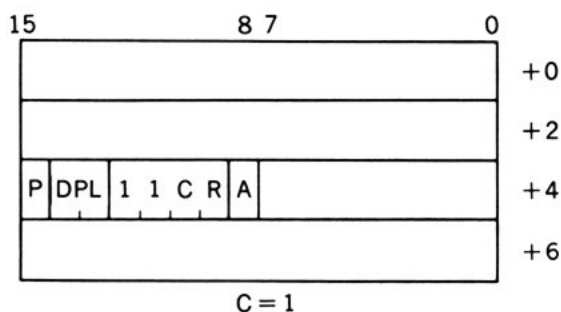



図 4·18 コンフォーミングコードセグメントのアクセスライト

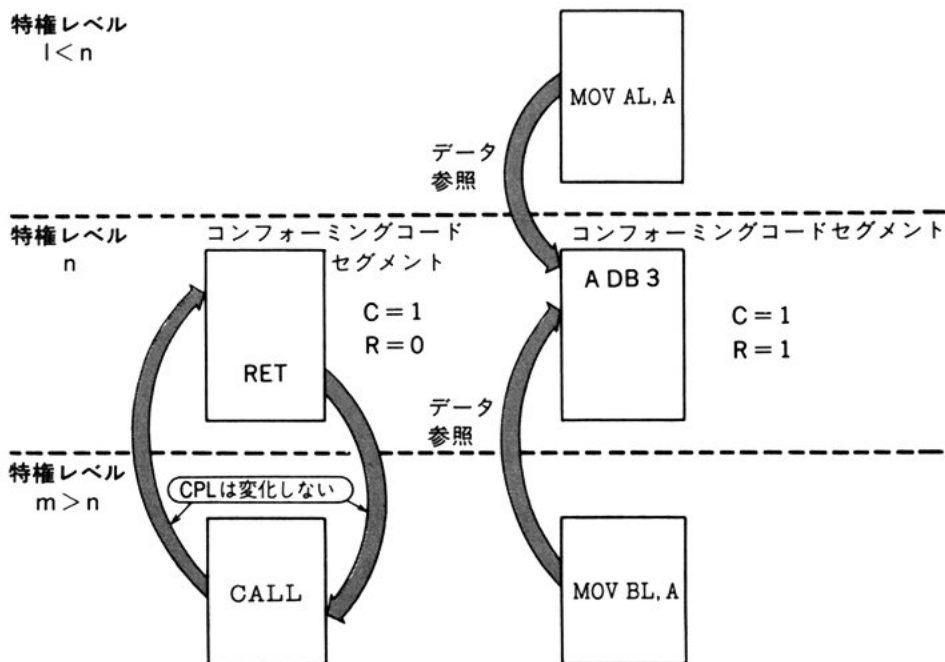



図 4·19 例外特権規則

コンフォーミングコードセグメントの DPL ≦ CPL

の条件が満足されればコールゲートを使用する必要はない．すなわち，図4·19に示すようにコンフォーミングコードセグメント内部の手続き，ラベルには，特権レベルの低いセグメントからであれば，どこからでもCALL命令あるいはJMP命令を使用して制御を移行することができる．ただし，コンフォーミングコードセグメントに制御を移行しても，CPLは変化しない．CALL命令，JMP命令を実行したとき，CPLが3であれば，DPL＝0のコンフォーミングコードセグメントで定義された処理を実行する間も，CPLは3のままである．したがって，コンフォーミングコードセグメント以外の，特権レベルの高いセグメントを参照することはもちろんできない．

さらに，コンフォーミングコードセグメントのアクセスライトのRが1で，リード可能となっている場合，このセグメント内部のデータはすべて，任意の特権レベルからリードすることができる．

ASM 86 と ASM 286

8086から80386までの命令は上位互換性がはかられている．8086，80186のアセンブラはASM 86である．80186では，8086の命令に加えて10種類の新しい命令が含まれるが，この場合はASM 86を実行するコマンドにMOD 186のオプションスイッチを指定する．ASM 286は80286のオブジェクトプログラムを作成するアセンブラである．

オブジェクトプログラムはインテルで定義されたフォーマットでファイルに格納される．ASM 86，ASM 286も含めて一般にアセンブラが作るオブジェクトファイル化は再配置形式である．再配置形式とは，プログラムをメモリの任意のアドレスに配置できるように，プログラムを配置するアドレスによって影響される部分を変更可能にしたものである．

再配置形式オブジェクトファイルは，プログラムを配置するメモリのアドレスを決め，部分的な再編集をするまで実行することはできない．これらの作業は一般にリンカ，ローケータと呼ばれるユーティリティプログラムによって実行される．

4-8 「トロイの木馬」問題

〔1〕 **トロイの木馬**　システムの信頼性にとって，セグメントレジスタに代入されるセレクタは重要な鍵となる．80286は特権レベルによって，セグメントレジスタに代入する前にセレクタの値を検査し，ユーザプログラムがOSのデータを勝手に読んだり，書き換えたりしないように保護することができる．しかし，セレクタを提供するプログラムの特権レベルと，そのセレクタをセグメントレジスタに代入するプログラムの特権レベルが異なるとき，この保護は完全ではなくなる．

図4･20に簡単な例を示す．OSが特権レベル0に定義され，特権レベル3のユーザプログラムはOS内部で定義された手続きCOPYを，コールゲートCOPY_GATEを介して使用することができる．手続きCOPYはパラメータとして，2つのメモリ領域のポインタと転送するバイト数を受け取り，メモリからメモリへ指定されたバイト数だけのデータを転送する．この簡略化OSのプログラムリストを図4･21に示す．

手続きCOPYを使用して，転送処理を実行するユーザプログラムのリストを図4･22に示す．このプログラムは，ソース（転送もと）メモリのセレクタ，オフセット，デスティネーション（転送先）メモリのセレクタ，オフセット，そして転

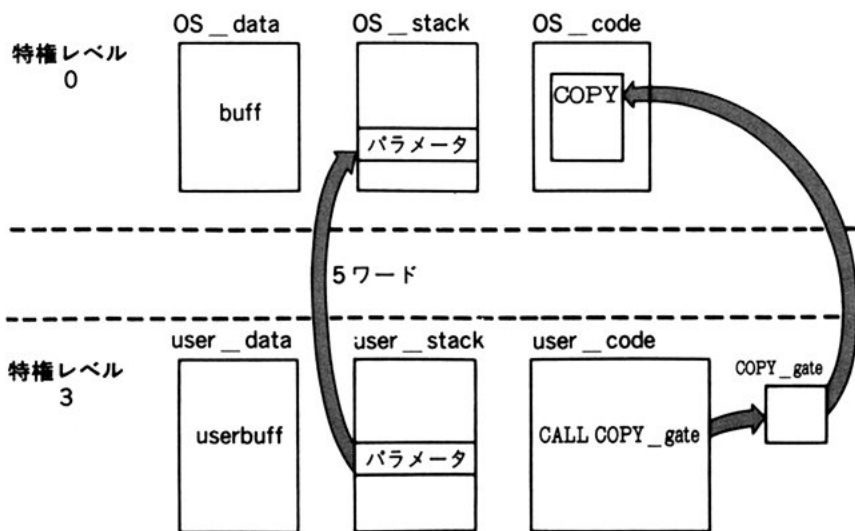



図 4・20　トロイの木馬

```
NAME simplified_os

PUBLIC copy

  os_data SEGMENT RW
     buff DB 'THIS DATA SHOULD BE PREVENTED FROM THE OUTER LEVEL.'
  os_data ENDS

  os_stack STACKSEG 100H

  os_code SEGMENT EO
          ASSUME DS:os_data
          ASSUME SS:os_stack
     stack_format STRUC
              old_bp      DW ?
              return_ptr  DD ?
              count       DW ?
              destin_ptr  DD ?
              source_ptr  DD ?
     stack_format ENDS

     copy PROC FAR
         PUSH BP
         MOV  BP,SP
         PUSH DS
         PUSH ES
         PUSH SI
         PUSH DI
         PUSH CX

         LDS SI,[BP].source_ptr
         LES DI,[BP].destin_ptr
         MOV CX,[BP].count
         CLD
      REP MOVSB


         POP CX
         POP DI
         POP SI
         POP ES
         POP DS
         POP BP
         RET 10
      copy ENDP
  os_code ENDS
END
```

➡ ASSUME 宣言はセグメント OS_data 内で定義した変数は DS を使用して参照することをアセンブラ ASM286 に指定する疑似命令である.

➡ 上と同様に, OS_stack 内で定義した変数は SS を使用して参照することを ASM286 に指定する. ASM286 は ASSUME 宣言に従って自動的にセグメントオーバライドプリフィックスを作るので, プログラマが各命令個々に指定する必要はない.

図 4・21 簡略化 OS の定義

```
NAME userprogram

EXTRN copy_gate:FAR

user_data SEGMENT RW
   userbuff DB 100H DUP(?)

user_data ENDS

user_stack STACKSEG 100H

user_code SEGMENT EO
          ASSUME DS:user_data
          ASSUME SS:user_stack

  start:
        PUSH 11000B        ➡ 転送するデータが定義されているメモリのセグメント
                             セレクタをPUSHする．
        PUSH 0             ➡ 転送するデータが定義されているメモリのオフセット
                             をPUSHする．
        PUSH user_data     ➡ 転送先のメモリのセグメントセレクタをPUSHする．
        PUSH 0             ➡ 転送先のメモリのオフセットをPUSHする．
        PUSH 51            ➡ 転送データのバイト数をPUSHする．
        CALL copy_gate

        JMP start
user_code ENDS
END start,DS:user_data,SS:user_stack
```

図 4・22 転送処理

送データのバイト数の順にスタックにPUSHしてから，コールゲートCOPY_gateをCALLする．ここで，ソースメモリのセレクタ11000BがOS内部で定義されたデータセグメントOS_dataのセレクタだとしたらどうなるだろうか．このプログラムにおいて，保護のための例外割り込みは発生せず，OS内部の重要なデータをユーザのメモリ領域にコピーしてしまう．なぜならば，セレクタ11000BをDSに代入するのは，CPLが0で動作する手続きCOPY自身だからである．ユーザプログラムは，11000Bをデスティネーションのセレクタとして COPY に渡せば，OSのデータを書き換えることもできる．このように，保護された処理の中に，パラ

```
ARPL   OP1,OP2

OP1 = { ワード汎用レジスタ
      { ワードメモリ

OP2 = ワード汎用レジスタ

OP1 [                    | RPL ]

OP2 [                    | RPL ]

if OP1.RPL < OP2.RPL
then OP1.RPL ← OP2.RPL, ZF ← 1
else ZF ← 0
```

図 4・23 ARPL命令とその機能

メータとして送られたデータがシステムに害を及ぼすことを，俗に「トロイの木馬」と呼ぶ．

〔2〕 ARPL命令による「トロイの木馬」問題の解決　「トロイの木馬」の問題は，セレクタを提供するプログラムとそれをセグメントレジスタに代入するプログラムの特権レベルが異なることから発生する．この問題を解決するために，80286はセレクタを提供するプログラムの特権レベルRPLについても検査を行う．また，ARPL命令を使用して，セレクタのRPLを，そのセレクタを提供したプログラムの特権レベルに一致させることができる．ARPL命令は図4·23に示すように，オペランドに2つのセレクタを指定し，もし，OP 1の下位2ビットがOP 2の下位2ビットより数値的に小さければ，OP 1の下位2ビットをOP 2の下位2ビットと同じ値にし，ZFを1にする．逆に，OP 1の下位2ビットがOP 2の下位2ビットより数値的に大きいか，または等しければ，ZFを0にして，OP 1の値は変わらない．

図 4·24 スタックセグメントとARPLの利用

ARPL命令は図4·24に示すように使用する．CALL命令が実行された直後，手続きの先頭においてBPをPUSHし，SPの値をBPに代入すれば，戻りCSの値は[BP+4]，問題のセレクタは[BP+14]で参照される．戻りCSの下位2ビットが，セレクタ11000Bを提供したプログラムの特権レベルである．したがって，ARPL命令の左のオペランドに[BP+14]，右のオペランドに[BP+4]の値を指定することによって，セレクタの提供者がその特権以上の特権レベルのセレクタを送っていないかどうかを検査することができる．このとき，戻りCSの下位2ビットは11Bであるから，ARPL命令によって，[BP+14]の値は11011Bに書き換えられる．この後，DSにセレクタ11011Bを代入しようとすれば，DPL ≧ MAX(CPL, RPL)の規則によって例外割り込みが発生する．また，例外割り込みを発生させるような命令を実行する前に，ZFを調べ

```
NAME simplified_os

PUBLIC copy

  os_data SEGMENT RW
     buff DB 'THIS DATA SHOULD BE PREVENT FROM THE OUTER LEVEL.'






  os_data ENDS

  os_stack STACKSEG 100H

  os_code SEGMENT ER
          ASSUME DS:os_data
          ASSUME SS:os_stack

     message DB 'YOU CANNOT READ MY DATA!!!'




     stack_format STRUC
               old_bp     DW ?
               return_ptr DD ?
               count      DW ?
               destin_ptr DD ?
               source_ptr DD ?
     stack_format ENDS

     copy PROC FAR
         PUSH BP
         MOV  BP,SP
         PUSH DS
         PUSH ES
         PUSH SI
         PUSH DI
         PUSH CX

         MOV AX,WORD PTR [BP].return_ptr+2
         ARPL WORD PTR [BP].source_ptr+2,AX
         JZ error

         LDS SI,[BP].source_ptr
         LES DI,[BP].destin_ptr
         MOV CX,[BP].count
         CLD
```

```
      REP MOVSB

          JMP exit

    error:
          MOV AX,SEG message
          MOV DS,AX
          MOV SI,OFFSET message
          LES DI,[BP].destin_ptr
          MOV CX,LENGTH message
          CLD
       REP MOVSB


    exit:

          POP CX
          POP DI
          POP SI
          POP ES
          POP DS
          POP BP
          RET 10
       copy ENDP
  os_code ENDS
END
```

図 4・25 「トロイの木馬」問題の解決

て条件JMP命令で処理の方向を変えることができる．ARPL命令を用いて「トロイの木馬」の問題を解決した手続きCOPYのプログラムリストを図4・25に示す．

4-9 I/O参照の保護

80286は16Mバイトのメモリ空間とともに，64KバイトのI/O空間をもつ．このI/O空間についても，メモリのセグメントと同様に特権による保護が実行される．I/O空間の特権レベルは図4·26に示すようにFLAGのIOPLの2ビットに定義され，これはセグメントディスクリプタのDPLに相当する．IN命令，OUT命令を初めとして，I/O空間を参照する命令を実行するときは

IOPL ≧ CPL

の条件を満足しなければならない．たとえば，IOPL＝0であるとき，CPL＝3の状態でIN命令を実行すれば，80286はタイプ13の割り込みを発生する．

このように，ユーザプログラムには直接的なI/O参照を禁止し，実際のI/O参照をOSレベルの処理で実行するようにシステムを作ることもできる．

I/O空間を参照する命令において
IOPL≧CPL
の条件を満足しなければならない．

図 4·26 I/O空間の特権レベル

5. 割り込み処理

プログラムとは独立な原因によって処理の流れを変更することを**割り込み**と呼ぶ．また，割り込みによって実行される手続きを**割り込み手続き**と呼ぶ．割り込みは周辺装置のサービス，CPU 内部または外部で発生する例外処理，デバッグなどに利用される．80286 は一度に 256 種類までの割り込み手続きを定義することができ，それぞれの割り込み手続きを 0 から 255 までのタイプ番号で識別する．リアルモードにおける割り込みは，いくつかの新しい内部割り込みが定義されている以外は，8086 のものと同じである．ここでは，主にプロテクトモードにおける割り込みについて述べる．

5-1 割り込みの原因

割り込みが発生する原因は，**ハードウェア割り込み**，**内部割り込み**，**ソフトウェア割り込み**の3種類に分類することができる．ハードウェア割り込みは，80286に外部から割り込み信号を与えることによって，割り込みを発生させるものである．図5･1に示すように，80286は3種類の割り込み信号入力端子をもつ．NMI端子はハードウェアシステムにとって致命的な現象を発見するために使用する．たとえば，メモリにパリティーエラーを検査する回路を組み込んでおき，パリティーエラーが発生した場合は，NMI端子にトリガ信号を供給するようにする．このとき，80286は無条件にタイプ2の割り込みを発生する．

周辺装置などからの割り込み信号は，INTR端子に入力する．INTR端子への割り込み信号は制御フラグのIFによってマスクすることができる．IFが1であるときに，INTR端子にHighの信号が入力されれば，80286は**割り込みアクノリッジサイクル**と呼ぶ一種のリードバスサイクルを実行し，外部の割り込み制御回路から1バイトの割り込みタイプを入力して，割り込みを発生させる．しかし，IFを0にしておけば，80286はINTR端子の信号を無視する．80286の割り込み制御回路には，8086と同様に**8259 A**を使用する．

$\overline{\text{ERROR}}$端子は数値演算プロセッサ80287からの例外割り込みに使用する．

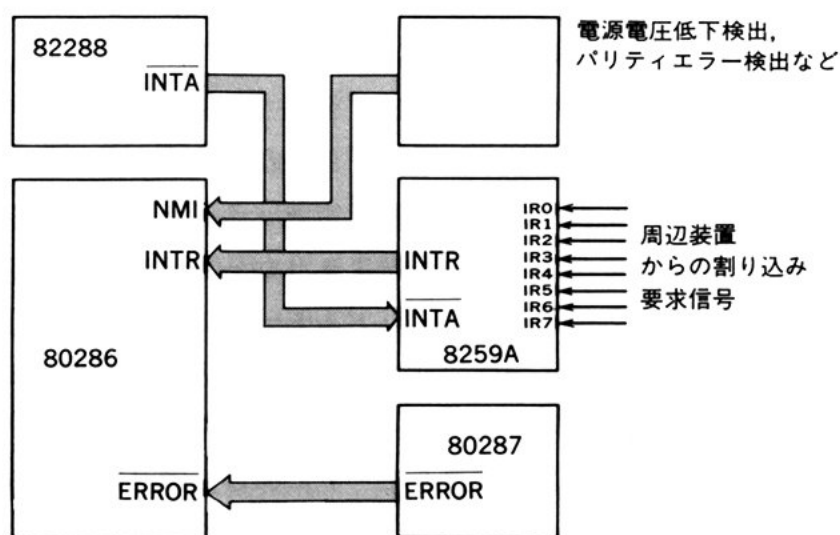



図 5･1 ハードウェア割り込み

80287 については第 9 章において述べる．

内部割り込みとは，80286 が例外処理の必要を認めたとき，内部で自動的に発生する割り込みであり，表 5·1 に示す割り込みが定義されている．

ソフトウェア割り込みは，INT 命令などの命令によって割り込み処理を実行する．表 5·2 にソフトウェア割り込みの命令を示す．

表 5·1 80286 内部割り込み

割り込みタイプ	割り込みの原因
0	DIV，IDIV 命令において，結果の商が AL または AX で表現できない大きさになったとき．
1	TF ビットが 1 のとき，1 命令を実行した後発生する．デバッグのシングルステップに利用できる．
6	不正命令コードの実行．80286 が定義されていない命令コードを実行しようとしたとき発生する．
7	80287 が存在しないとき（EM＝1 かつ MP＝0），ESC 命令が実行された．または 80287 が存在するとき（EM＝0, MP＝1），タスクスイッチ直後に（TS＝1）ESC 命令または WAIT 命令が実行された．
8	2 重例外割り込みの場合．
9	80287 オーバランが発生した．すなわち，80287 が扱う 1 ワード以上のデータの一部がセグメントリミットを越えたときに発生する．
10	不正な TSS が参照された．
11	セグメントが存在しない（P＝0）．
12	スタックセグメントに関する例外割り込みの場合．
13	一般プロテクトエラーが発生した場合の例外割り込みの場合．

表 5·2 ソフトウェア割り込み

割り込み命令	発生する割り込みタイプ
INTn（n=0~255）	0~255
INTO	4
INT 3	3
BOUND	5

5-2 割り込みのプロセスとIDT

〔1〕 **割り込み処理とIDT**　5·1で述べた原因によって割り込みが発生するが，80286は各命令の実行終了時に割り込みの発生があるかどうかを認識する．しかし，例外として，MOV命令，POP命令のようにSSに値を代入する命令の後では，外部からのハードウェア割り込みが発生していてもそれを無視する．その次の命令の実行終了時に割り込み処理を実行する．これは，SSだけを変更して，SPを未変更の状態で割り込み処理を実行したとすれば，誤ったメモリ領域をスタックとして使用することになるからである．したがって，SSを変更する命令を実行した直後に，SPを変更する命令が必ず実行できるようになっている．

80286が割り込み発生を認識してから，割り込み手続きを実行するまでに，図5·2に示すような手順を実行する．このように割り込み手続きの最初の命令を実行する前に，自動的に実行される処理を**割り込み処理**と呼ぶことにする．割り込みタイプは，すでに定義されているもの，INT命令のオペランドで指定するもの，割り込みアクノリッジサイクルを実行して，外部からリードするものがある．割り込みタイプが決まれば，図5·3に示す**IDT**（interrupt descriptor table）と呼ばれる特別なテーブルのオフセットが［割り込みタイプ×8］の領域から6バイトのデータを読む．IDTは，プロテクトモードにおける割り込みベクトルテーブルであり，IDTのベースアドレスとサイズはIDTRによって定義される．IDTとIDTRの関係は，GDTとGDTRの関係とまったく同じであり，LGDT命令，SGDT命令によって，メモリからGDTRに値を代入したり，また，GDTRの値をメモリに書いたりできるように，IDTRについても，LIDT命令，SIDT命令を使用して，メモリからIDTRに値を代入したり，IDTRの値をメモリに書いたりすることができる．

〔2〕 **割り込みゲート，トラップゲートによる制御移行**　しかし，IDTに定義するディスクリプタは，図5·4に示すような**割り込みゲート**または**トラップゲート**の2種類のゲートである．これらのゲートは，ワードカウントを指定しないことを除けば，コールゲートとよく似ていて，割り込みによって実行される手続きの実行開始アドレスを定義する．割り込みゲートとトラップゲートの使い分けは，ただ1つ，割り込み処理においてIFを0にするか，しないかだけである．

割り込み手続きの実行中に，INTR 端子による割り込みを無視したいときは，IF を 0 にするために割り込みゲートを使用する．逆に，割り込み手続きの実行中でも，INTR 端子からの新たな割り込み要求を受ける場合は，IF の値を変化させないトラップゲートを使用する．

割り込み処理において，FLAG，戻りアドレスのセグメントセレクタ，戻りアドレスのオフセットの順に，自動的にスタックに PUSH される．さらに，例外

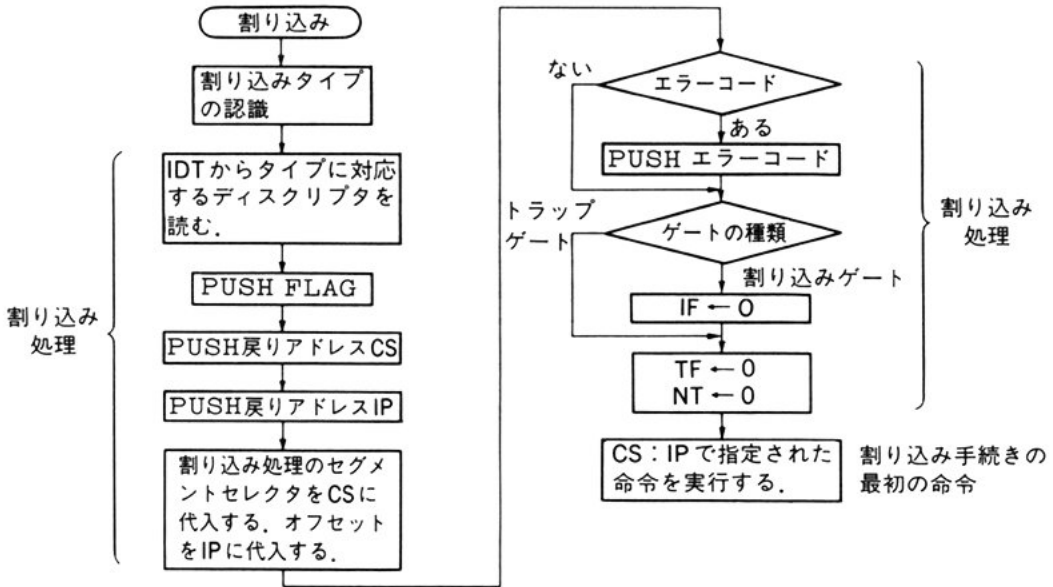



図 5・2 割り込みの手順

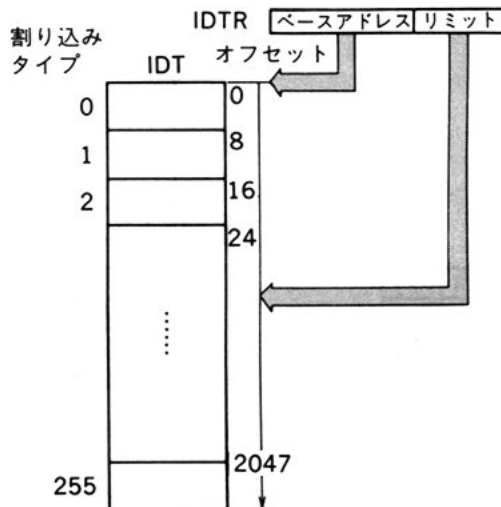
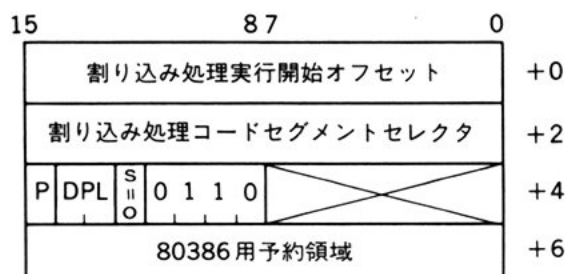
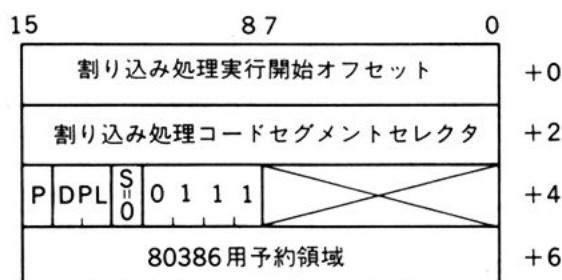



図 5・3 IDT と IDTR

(a) 割り込みゲート

(b) トラップゲート

図 5・4 割り込みゲートとトラップゲート

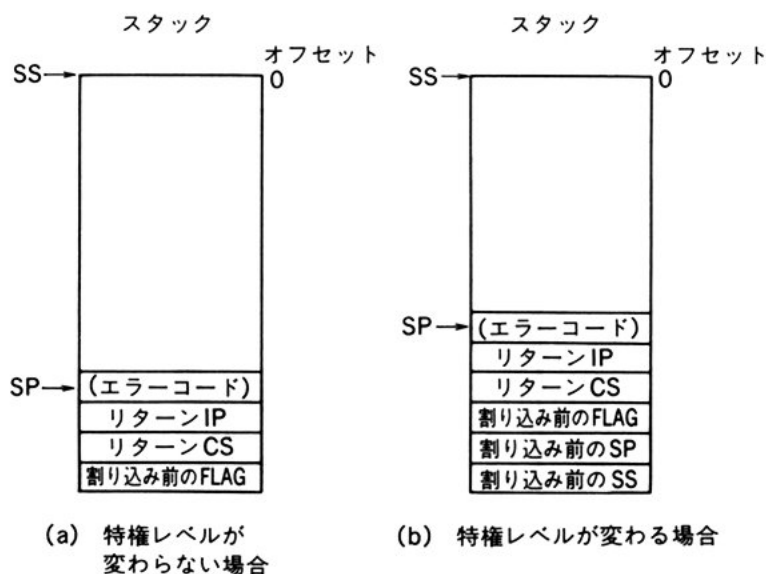
(a) 特権レベルが変わらない場合　(b) 特権レベルが変わる場合

注) 割り込みの原因によって，エラーコードをPUSHする場合と，しない場合がある．

図 5・5 割り込み発生後のスタック

割り込みの中には，エラーコードを PUSH するものがある．したがって，割り込み発生後のスタックは，図 5·5 に示すようになる．

また，割り込みゲート，トラップゲートを使用して，割り込み手続きに制御を移行する場合も，コールゲートを参照する CALL 命令の場合と同じ規則に従って，特権レベルを変更することができる．特権レベルが変わるときは，やはりスタックセグメントも自動的に更新される．特権レベルが変わったときの新しいスタックの様子を図 5·5 (b) に示す．

〔3〕 **割込み処理の優先**　80286 が割り込みを認識したとき，2つ以上の割り込みが同時に発生している場合は，表 5·3 に示す順序で処理される．たとえば，NMI 端子へのトリガ入力によって，タイプ2の割り込み要求と DIV 命令の実行によって，タイプ0の割り込みが同時に発生しているときは，タイプ0の割り込み処理を先に実行する．しかし，80286 は割り込み手続きを実行する前に再び，割り込みの発生を認識するので，タイプ0の割り込み手続きを実行する前に，タイプ2の割り込み処理を実行する．したがって，タイプ2の NMI の割り込み手続きの方が先に実行される．

〔4〕 **割込み手続きの定義**　以上のような割り込み処理を実行した後，初めて割り込み手続きの最初の命令が実行される．

割り込み手続きの定義は CALL 命令で引用される手続きと変わるところはない．ただし，割り込み手続きに制御を移行する割り込みゲートまたはトラップゲートを IDT に定義すればよい．また，割り込み手続きから元のプログラムに制

表 5·3 割り込み処理の順序

割り込みを処理する順序	割り込みの種数
1	命令実行時に内部で発生する例外割り込み
2	TF を 1 に設定したときに発生するシングルステップ
3	NMI
4	タイプ 9 の 80287 オーバーラン
5	INTR

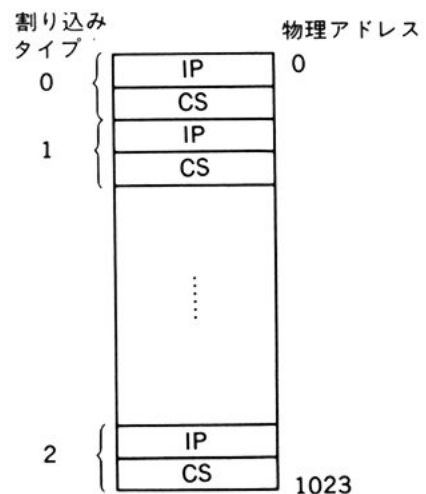



図 5·6 リアルモードにおける IDT

御を戻すためには IRET 命令を使用する．IRET 命令はスタックから戻りアドレスの IP，CS を順に POP し，最後に FLAG を POP して割り込みの入った元のプログラムに制御を移行する．また，割り込みによる制御移行で特権レベルが変化している場合には，RET 命令と同じようにスタックを切り換えて低い特権のプログラムに戻ることができる．

また，80286 は DIV 命令，BOUND 命令など，命令を実行中に発生した例外割り込みについては，割り込みの原因となった命令の先頭アドレスのセグメントとオフセットをスタックに PUSH するから，IRET 命令を使用するときに注意する必要がある．すなわち，割り込みの原因を取り除いてからそのまま IRET 命令を実行して，前に割り込みの原因となった命令を再実行するか，またはスタックに保存されている戻りアドレスの IP を書き換えてから IRET 命令を実行して割り込みの原因となった命令をスキップして，次の命令に戻るようにする．

〔5〕 **リアルモードの IDT**　なお，リアルモードでの IDT は 8086 の割り込みベクトルテーブルとまったく同じ状態になっている．すなわち，図 5･6 に示すように物理アドレスの 0 番地から 1023 番地までに，割り込みタイプ 0 から 255 までのそれぞれの割り込み手続きの実行開始アドレスだけが定義されたものである．リアルモードにおいても，LIDT 命令を使用し，IDTR の値を変更することによって，IDT の定義アドレスを変更することができるが，これはリアルモードからプロテクトモードに移行するときの初期設定として実行するべきものであり，リアルモードで動作するシステムにおいては絶対に使用してはならない．

5-3 ハードウェア割り込み

ハードウェア割り込みと呼ばれるものには，**5·1** で述べたように NMI，INTR，$\overline{\text{ERROR}}$ の各端子から入力される割り込み信号が原因となって発生するものがある．ここでは，NMI，INTR 端子に入力される信号の形式について述べる．

NMI はメモリパリティエラー，電源電圧低下など，システムにとって致命的な状況に対する例外処理を実行するために使用される．NMI 端子には図 5·7 に示すような，4 クロック以上の間 Low を保ち，High に立ち上がった後，4 クロック以上の間，High を保持する信号を入力する．NMI 信号は 80286 内部でラッチされ，タイプ 2 の割り込みを発生する．NMI の割り込み処理を実行中に，再び NMI 割り込み信号が入力されると，その信号は 80286 内部でラッチされるが，IRET 命令を実行するまで処理されない．IRET 命令を実行した後に，初めてラッチされていた NMI 割り込みが処理される．

INTR 端子はレベルセンスの端子であり，各命令の終了時に INTR 端子の電圧レベルを検査し，High であれば，割り込みアクノリッジサイクルと呼ぶ一種のリードバスサイクルを実行し，割り込み制御回路 8259 A から 1 バイトの割り込みタイプをリードする．INTR 端子からの割り込み要求は IF によってマスクすることができる．

80286 が実行する割り込みアクノリッジサイクルを図 5·8 に示す．

また，80286 と 8259 A の接続の様子を図 5·9 に示す．8259 A の使用は基本的には 8086 の場合と同じであるが，図 5·8 に示すように，割り込みアクノリッジサイクルのサイクル 1，サイクル 2 それぞれに少なくとも 1 つのウェイトサイクルを入れなければならないことに注意する．このウェイトサイクルは一般のバスサイクルと同様に $\overline{\text{READY}}$ によって制御する．

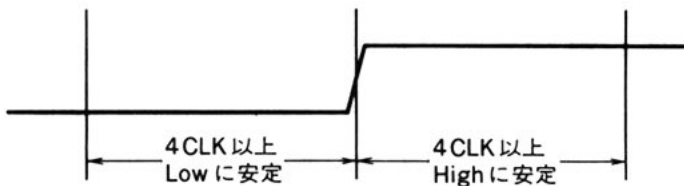



図 5・7 NMI 割り込み信号

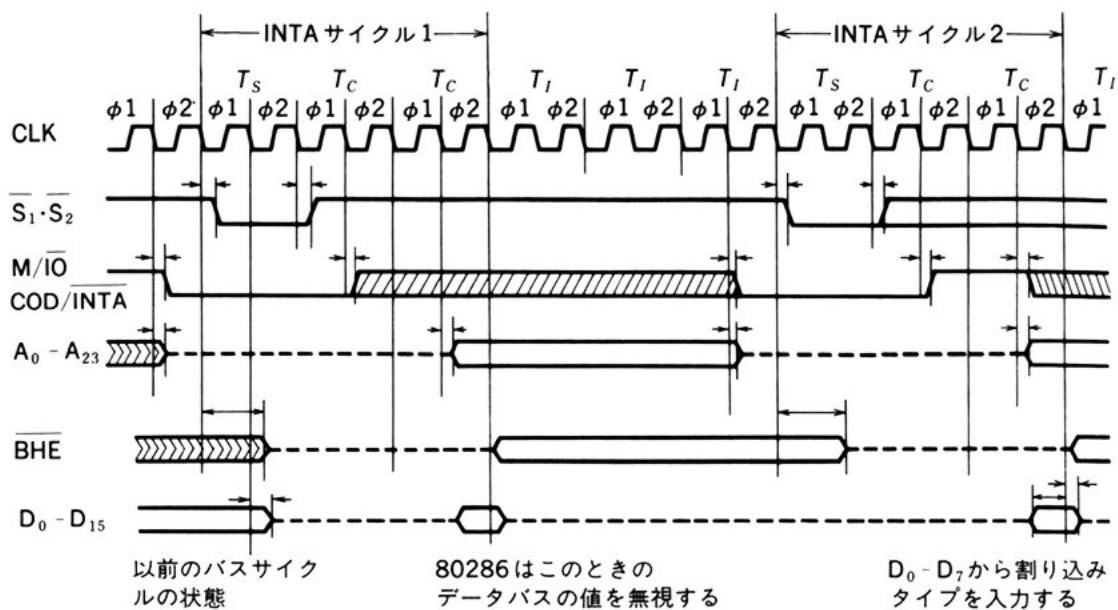



図 5・8 80286割り込みアクノリッジサイクル

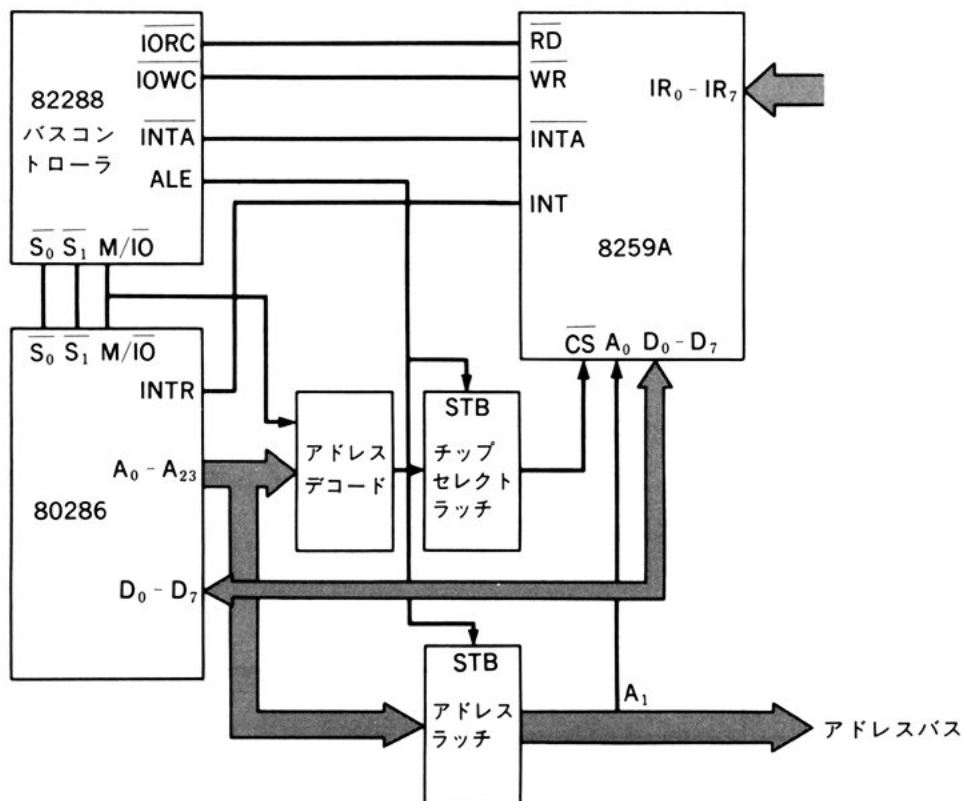



図 5・9 80286と8259Aの接続

5-4 ソフトウェア割り込み

表5·2に示したソフトウェア割り込みは，INT命令などのソフトウェア命令によって割り込み処理を実行するもので，CALL命令によって手続きに制御を移行させるのとよく似ている．ソフトウェア割り込みがCALL命令と異なるところは，スタックにFLAG，戻りアドレスのCSとIPをPUSHして，IDTに定義された割り込みゲートまたはトラップゲートを介して，制御を移行させるこ

図5・10 INTO命令の利用

とである．したがって，INT 命令はシステム内部のどの応用プログラムからでも引用されるような最も基本的な手続を引用するときに便利である．

〔1〕 **INTO 命令**　INTO 命令は INT 命令の中で特別な命令である．これは OF が 1 のとき，タイプ 4 の割り込みを発生するが，OF が 0 のときは，そのまま次の命令を実行する．INTEGER タイプのデータの演算の後に INTO 命令を書くことによって，整数演算でオーバフローが発生したときの例外処理をタイプ 4 の割り込みによって実行することができる．図 5·10 に，整数変数 I1 に 5 を加算した後，オーバフローが発生すれば，タイプ 4 の割り込み処理によって，"**OVER FLOW ERROR**" のメッセージを CRT に表示するプログラム例を示す．もちろん，このとき IDT のオフセット 32 から 8 バイトの領域に，手続き INT4 に制御を移行するための割り込みゲートまたはトラップゲートを定義しておかなければならない．

〔2〕 **プログラムデバッグに利用する割り込み**　INT n 命令は 2 バイトの命令であるが，INT 3 だけは 1 バイトの命令である．これは，プログラムのデバッグにおいて，実行を中断するブレイクポイントを指定するために使用する．すなわち，プログラムで中断したい部分の命令を INT 3 命令に置き換えておいて，プログラムを実行すれば中断したいところでタイプ 3 の割り込みに入る．タイプ 3 の割り込みが発生したとき，INT 3 命令を元の命令で，再び置き換えておけばよい．

プログラムのデバッグにおいて，1 命令ずつ実行したい場合は，制御フラグの TF を利用することができる．TF を 1 にセットしておけば，80286 は 1 命令を実行した後，タイプ 1 の割り込みを発生する．

ここで，TF だけを 1 にする命令はないので，実際の処理では図 5·11 に示すように，スタックに FLAG，実行する命令の CS, IP を PUSH した後で IRET を実行することによって，FLAG を書き換えて目的の命令を実行することができる．もちろん，IRET 命令を実行する前に，スタックに入っている FLAG の TF を 1 にしておく．この処理の流れを図 5·12 に示す．

〔3〕 **BOUND 命令**　BOUND 命令は配列のレンジチェックを行う命令で，図 5·13 に示すように使用する．配列を定義するとき，配列の最小と最大のオフセットを 4 バイトの変数 BUFFRANGE に定義しておけば

```
BOUND BX,BUFFRANGE
```

```
PUSHF
PUSH  実行する命令のセグメントセレクタ
PUSH  実行する命令のオフセット
MOV BP,SP
OR WORD PTR[BP+4],0000000100000000B
IRET
```

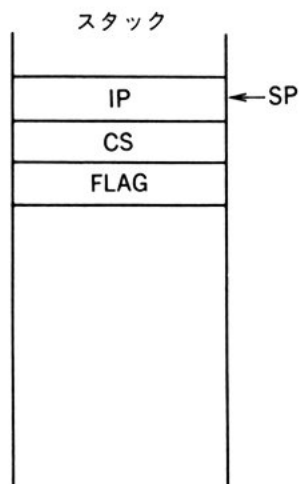



図 5・12　1命令だけを実行する方法

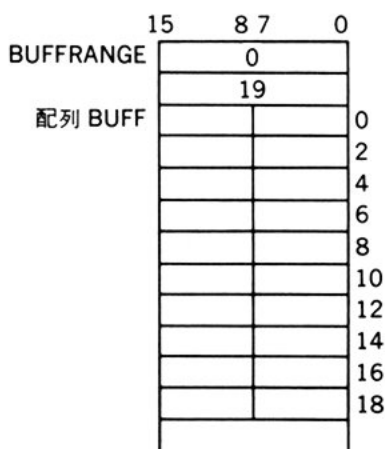
図 5・11　IRET 命令を実行する前のスタック

```
BOUND BX, BUFFRANGE
MOV BUFF [BX], 0FFH
```

図 5・13　BOUND 命令の利用

のような命令によって，BX の値が BUFFRANGE の下位ワードに定義された値より大きく，BUFFRANGE の上位ワードに定義された値より小さいかどうかを検査できる．図5・13 の場合，BX ≧ 0，かつ，BX ≦ 19 のとき，そのまま次の命令を実行するが，BX の値が 0 から 19 の範囲に入っていないとき，タイプ5 の割り込みを発生する．ここで，BOUND 命令の左のオペランドは，BX 以外にワード汎用レジスタであればなにを指定してもよい．

5-5 内部割り込み

命令の実行中に80286が自動的に発生する割り込みを内部割り込みと呼ぶ．内部割り込みの中で，保護例外と呼ばれるものについては第7章において述べる．ここでは，表5·4に示すような保護例外以外の内部割り込みについて述べる．

DIV命令，IDIV命令を実行したとき，商はバイト操作のときはAL，ワード操作のときはAXに残るが，0で割り算を実行した場合とか，商がAL，AXで表現できる値より大きい場合には，タイプ0の割り込みを発生する．

タイプ1の割り込みについては，**5-4**で述べたとおりである．

タイプ6の割り込みは，80286が定義されていない不正な命令コードを実行したときに発生する．

80286の内部割り込みの戻りアドレスは，タイプ1のシングルステップ以外は割り込みの原因となった命令の先頭アドレスがスタックにPUSHされることに注意する．たとえば，8086では，タイプ0の割り込みのとき，DIV命令またはIDIV命令の次の命令のアドレスが，戻りアドレスとしてスタックにPUSHされたのに対して，80286では，DIV命令またはIDIV命令自身の先頭アドレスが，戻りアドレスとしてスタックにPUSHされる．したがって，そのままIRET命令を実行すれば，割り込み発生の原因となった命令を再実行する．このことはタイプ6の不正命令コード実行の例外割り込みについても同じである．

表5·4 内部割り込み

割り込みタイプ	原因	戻りアドレス
0	DIV，IDIV命令実行時に結果の商が，AL，AXで表現できない．	割り込みの原因となった命令の先頭アドレス
1	TF＝1となっている場合，1命令実行後発生する．	次の命令のアドレス
6	80286で定義されていない．不正命令コードを実行しようとした．	割り込みの原因となった命令の先頭アドレス

6. タスクとタスクスイッチ

80286のプロテクトモードでの特徴は，厳密なメモリ管理とタスク管理である．8 MHzで動作する80286では，約23 μsでタスクスイッチを実行することができる．第6章では，タスクとは何か，80286はタスクをどのように管理するのか，そして，タスクスイッチの命令について述べる．

6-1 シングルタスク

シングルタスクシステムは図6·1に示すように，1つのプログラムがその実行開始から終了までCPUを専有するシステムである．たとえば，MS-DOS（V3.Xまでのもの）とかCP/M-86のようなOSを考えればよい．プログラムAを実行するとき，キーボードから入力されたコマンドによって，OSがディスクファイルからプログラムをメモリにロードし，レジスタに必要なデータを初期設定してから，制御をプログラムAに移行する．この後，次のプログラムを実行するためには，プログラムAが実行を終了するまで待たなければならない．

図6·1において，プログラムAが終了すると制御はプログラムBに移行する．ここで，プログラムBはOSのコマンド入力処理と考えればよい．OSは再びCRTにプロンプトマークを表示し，次のコマンドが入力されるのを待つ．

このようなシステムでは，一度に一人のユーザしかコンピュータを利用することができない．また，このようなシステムを機械制御に利用することを考えると，必要に応じて要求されたプログラムを十分なレスポンスで実行させることができない．

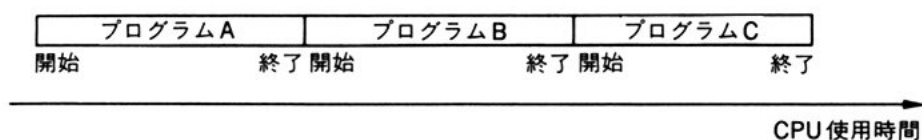



図 6·1　シングルタスクシステム

6-2 マルチタスク

〔1〕 **マルチタスクの動作**　**マルチタスク**とは2つ以上のプログラムを同時に実行でき，前のプログラムが実行を終了していなくても，次のプログラムの実行を開始できるシステムである．マルチタスクシステムの応用としては，**タイムシェアリングシステム**，**リアルタイムシステム**がある．もちろん，80286が本当の意味で，同時に2つ以上のプログラムを実行することはできない．限られた特定の時間を考えれば，80286は1つのプログラムを実行するだけである．ここで，マルチタスクシステムで実行されるプログラムのことを**タスク**と呼ぶ．プログラムとタスクとは1対1に対応しない．なぜならば，メモリにロードされた1つのプログラムが，2つの別々のタスクとして実行されることがあるからである．

図6･2に3つのタスクA, B, Cが実行される様子を示す．80286の実行時間は分割され，各タスクに分け与えられる．最初，80286はタスクAを実行するが，時間t_1からt_3の間は別のタスクを実行する．したがって，時間t_1でタスクAの処理を中断してから，時間t_3で再開するまでの間，レジスタの値などの80286の状態をメモリ上の特別なセグメントに保存する必要がある．この特別なセグメントを**TSS** (task status segment) と呼ぶ．

TSSはタスクごとに1つずつ定義する．時間t_1において，80286の使用権をタスクAからタスクBに渡すとき，現在のレジスタの状態をタスクAのTSSに保存してからタスクBのTSSに保存されているレジスタの状態を80286のレジスタに代入する．この後，80286が以前と同様に実行を続ければタスクAの実行が完全に終了していないにもかかわらず，80286はタスクAとは関係のない別のタスクを実行することができる．

このことは，時間t_2においてタスクBからタスクCに切り換わるときも同様である．時間t_3において，再びタスクAに80286の使用権が渡るとき，80286のレジスタの状態がタスクCのTSSに保存されてからタスクAのTSSから80286のレジスタに，時間t_1において保存された状態が再び代入される．

もし，図6･2に示すシステムにおける各タスクが，3人のユーザがそれぞれ3台のターミナルを使用して，別々に実行しているプログラムであるならば，それぞれのユーザは1つの80286を他のユーザと分け合っていることには気づかな

いだろう.

また，マルチタスクシステムを機械制御に利用するならば，新しい仕事の要求に対して，実時間（リアルタイム）で応答できるシステムを作ることができる．前者の例では，タイマを使用して一定時間ごとに必ずタスクを切り換えるようにする．このようなシステムを**タイムシェアリングシステム**と呼び，実行可能なタスクは一定時間ごとに必ずCPUのサービスを受ける．

また，後者のリアルタイムシステムでは，新しいデータが入力されるなどの何らかの事象の発生によってタスクを切り換える．もちろん，このようなシステムでは，実行可能なタスクの中で次にどのタスクにCPUを与えるかを決定し，管理する**タスク管理**のプログラムが必要である．

〔2〕 **80286 の TSS**　　TSS は一種のセグメントであるから，最大 64 K バイトの大きさまで定義することができるが，80286 ではそのオフセット 0 から 43 までの構造が図 6・3 に示すように決められている．また，TSS は図 6・4 に示す **TSS ディスクリプタ**によって定義され，TSS ディスクリプタのセレクタによって TSS を一意に識別することができる．なお，TSS ディスクリプタは必ず GDT に定義しなければならない．

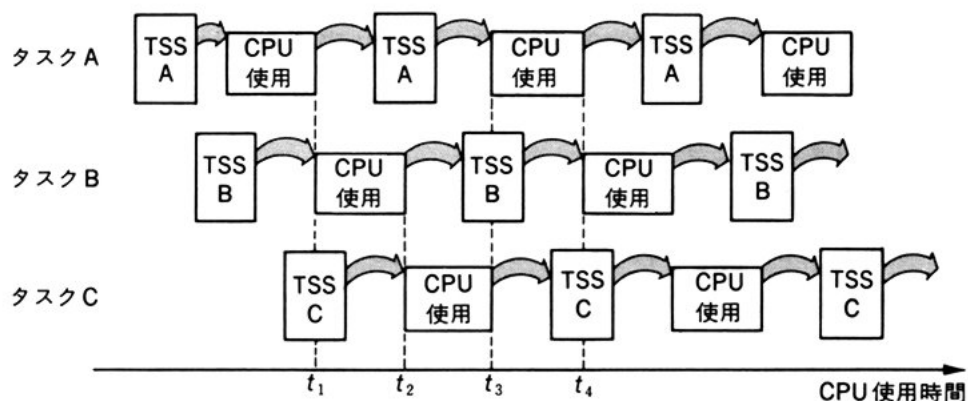
TSS ディスクリプタのベースアドレス，リミットには，TSS が定義されるベースアドレスと TSS のリミットを指定する．ここで，TSS のリミットは必ず 43 以上の値でなければならない．TSS ディスクリプタのアクセスライトの P，DPL も他のディスクリプタのものと同様である．

ここでTSS（＝Task Status Segment）は80286のレジスタの値などの内部状態を格納する特別なセグメント．

図 6・2　マルチタスクシステム

15　0

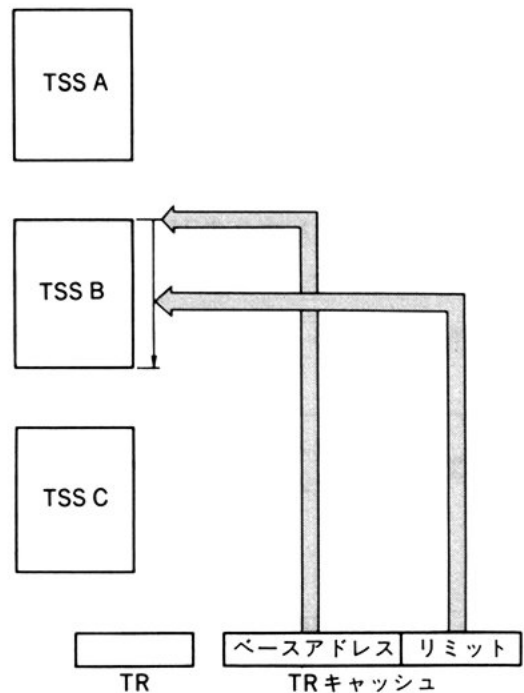
フィールド	オフセット
バックリンクセレクタ	0
特権レベル0 SP	2
特権レベル0 SS	4
特権レベル1 SP	6
特権レベル1 SS	8
特権レベル2 SP	10
特権レベル2 SS	12
IP(実行開始オフセット)	14
FLAG	16
AX	18
CX	20
DX	22
BX	24
SP	26
BP	28
SI	30
DI	32
ES	34
CS	36
SS	38
DS	40
LDTセレクタ	42

図 6・3　TSS
(Task Status Segment)

図 6・4　TSS ディスクリプタ

図 6・5　TR と TR キャッシュ

TSSディスクリプタを参照するときは(TSSディスクリプタを参照する命令は後に述べるようにCALL命令，JMP命令などである)，P＝1でなければならないし，またDPLに関して，DPL≧MAX(CPL, RPL)でなければならない．TSSディスクリプタのBは80286が自動的に1にセットしたり，また0にクリアするビットで，B＝1のときTSSディスクリプタの指定するタスクが処理中であることを表す．

TSSのオフセット44からの領域は，たとえば数値演算プロセッサ80287のレジスタ状態を保存するなど，必要に応じてユーザが任意に使用すればよい．TSSのオフセット14から41までの14ワードに，80286のすべてのレジスタの状態が保存される．TSSのオフセット2から13までは，特権レベルが変わったときに使用されるSS，SPの初期値を保存する．ここで，より低い特権レベルから特権レベル3に移行することはありえないから，特権レベル3のSS，SPの初期値を

記録する必要はない．オフセット42から始まる1ワードには，そのタスクで使用するLDTのLDTディスクリプタのセレクタを記録する．これによって，タスクごとに別々のLDTを指定することができる．最後に，オフセット0から始まる1ワードは，タスクのバックワードリンクで，以前に実行されていたタスクのTSSディスクリプタのセレクタを記録するために使用する．

〔3〕 **TSSの定義**　TSSは図6·5に示すように，タスクごとに1つずつ定義する．3つのタスクA，B，Cを実行するとき，3つのTSSを定義し，さらに3つのTSSディスクリプタをGDTに定義する．現在実行中のタスクのTSSは，TRとTRキャッシュによって定義される．TRとTRキャッシュの扱いは，セグメントレジスタとセグメントキャッシュの場合とよく似ている．図6·6に示すLTR命令を使用して，TSSディスクリプタのセレクタをTRに代入するとき，TSSディスクリプタのリミットとベースアドレスが自動的にTRキャッシュに代入される．このとき，GDTに定義されているTSSディスクリプタのBが自動的に1になる．また，TRの値はSTR命令によって読むこともできる．

LTR	｛ワードメモリ 　ワード汎用レジスタ｝	➡	ワードメモリまたはワード汎用レジスタに定義されたTSSセレクタをTRに代入する．このとき，自動的に対応するTSSディスクリプタの下位5バイトをTRキャッシュに代入する．
STR	｛ワードメモリ 　ワード汎用レジスタ｝	➡	TRのセレクタをワードメモリまたはワード汎用レジスタに代入する．

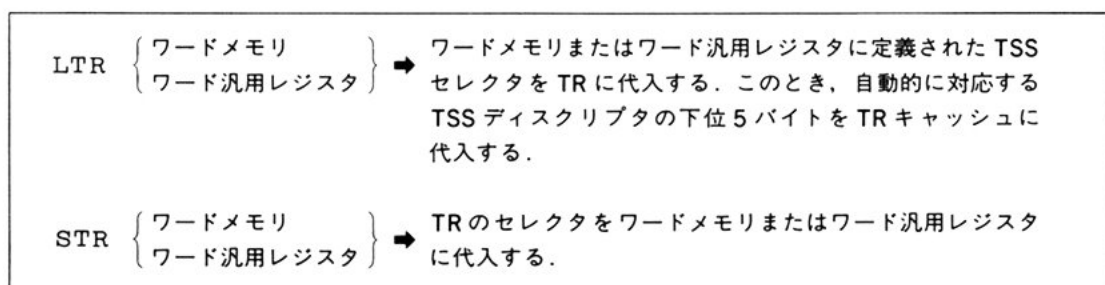



図6·6 TRに関連する命令

6-3 LDT と LDT ディスクリプタ

マルチタスクシステムにおいて，タスク間で共有するディスクリプタは GDT に定義し，特定のタスクだけで参照するディスクリプタは LDT に定義する．したがって，LDT はタスクごとに 1 つずつ定義し，タスクと LDT の対応を決めるために，図 6·7 に示すように，TSS のオフセット 42 から始まる 1 ワードに LDT ディスクリプタのセレクタを定義する．図 6·7 は 3 つのタスクの中でタスク B が実行されるときの，TR，TR キャッシュ，LDTR，LDTR キャッシュの

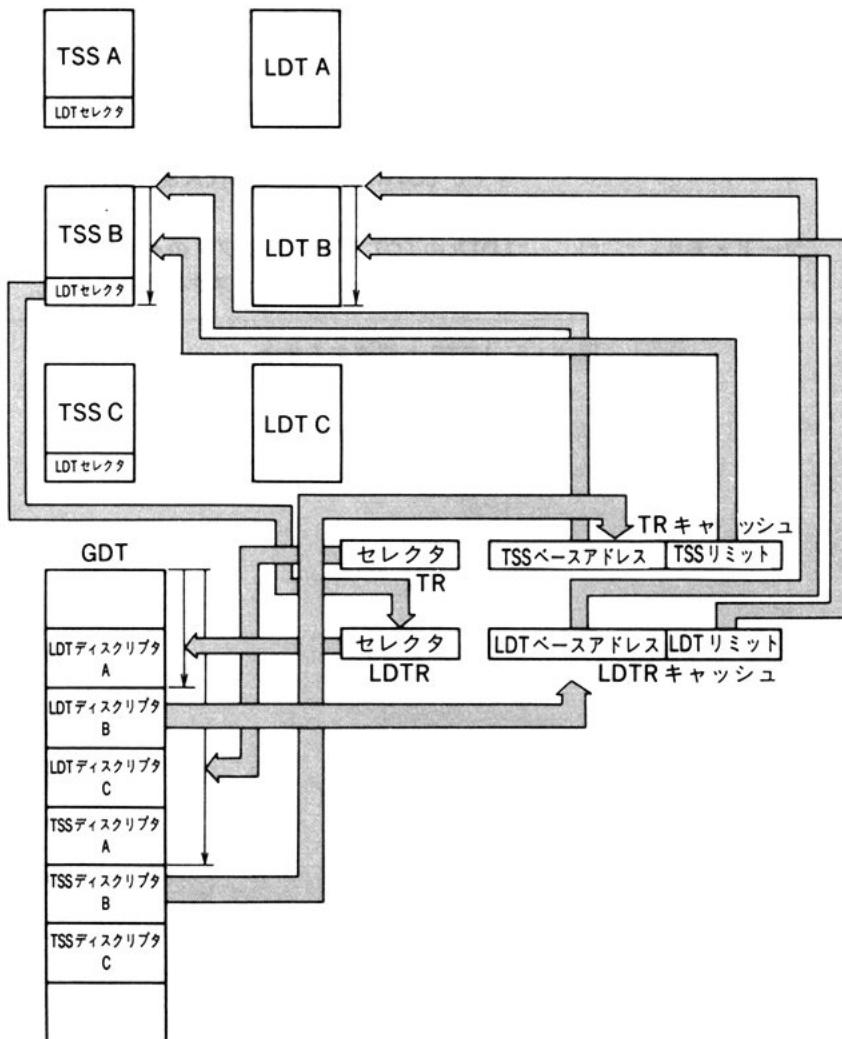



図 6・7　TSS と LDT

状態を示す．

現在実行中のタスクで使用されるLDTは，TSSを指定する方法と同様に，LDTR，LDTRキャッシュによって定義される．LDTR，LDTRキャッシュの扱いは，TR，TRキャッシュの場合とまったく同様である．図6･8に示すLLDT命令によって，LDTRにLDTディスクリプタのセレクタを代入するとき，自動的にLDTRのセレクタで指定されるLDTディスクリプタのリミットとベースアドレスがLDTRキャッシュに代入される．

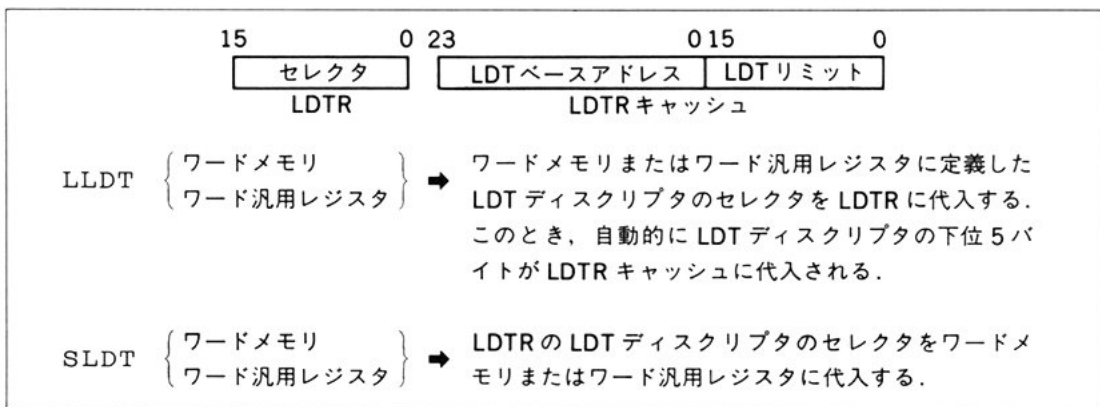



図 6・8　LDTRと関連する命令

6-4 タスクの定義

80286において，タスクを実行するまでには，図6·9に示すようにGDT，LDT，TSSをメモリに定義し，GDTR，LDTR，TRに必要な値を定義しなければならない．ただし，LDTRとTRは，セグメントレジスタと同じようにキャッシュをもち，GDTからキャッシュに自動的にディスクリプタが代入されるから，LDTディスクリプタ，TSSディスクリプタは必ずGDTに定義しておく必要がある．

GDTの定義から，1つのタスクを実行するまでの手順を以下に示す．

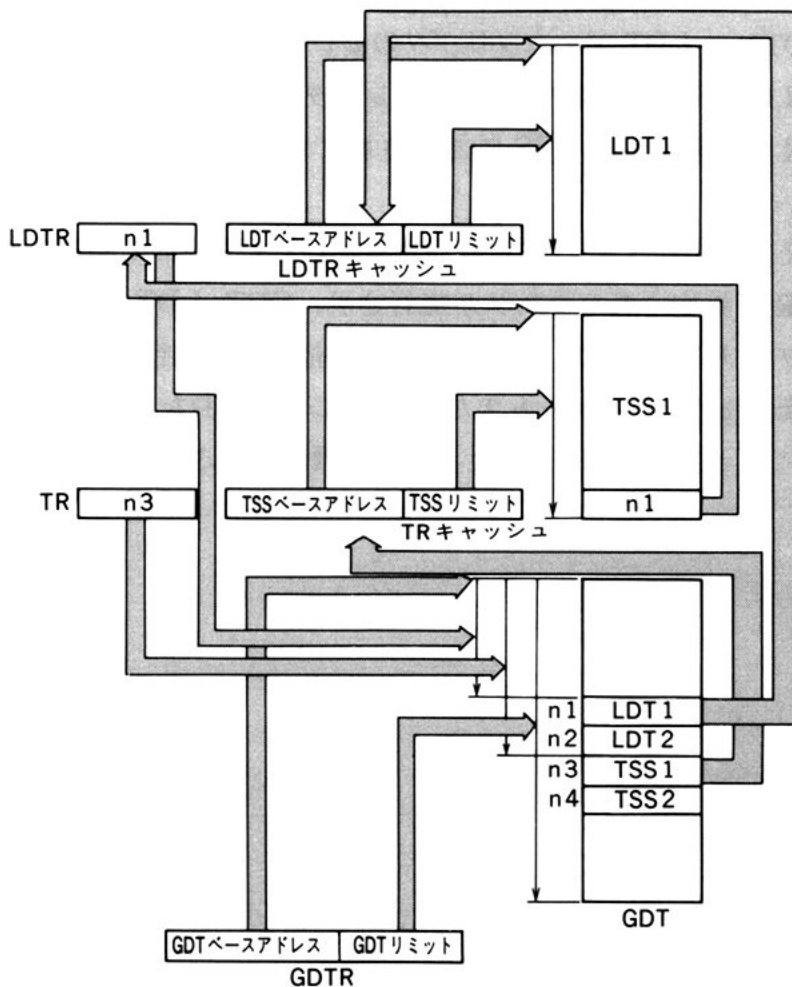



図 6·9 タスクの定義

（1）　メモリに GDT を書く．GDT には必要なセグメントディスクリプタ，ゲート，LDT ディスクリプタ，TSS ディスクリプタを定義する．

（2）　GDTR に GDT のベースアドレスとリミットを代入し，GDT を使用可能な状態にする．

（3）　メモリに IDT を定義する．IDT には割り込みゲートまたはトラップゲートを必要に応じて定義する．

（4）　IDTR に IDT のベースアドレスとリミットを代入し，IDT を使用可能な状態にする．

（5）　リアルモードからプロテクトモードに切り換える．

（6）　メモリにいくつかの TSS を定義する．

（7）　メモリにいくつかの LDT を定義する．

（8）　LDTR に LDT ディスクリプタのセレクタを代入する．

（9）　TR に TSS ディスクリプタのセレクタを代入する．

（10）　TR で指定された TSS に記録されているレジスタの状態を各レジスタに代入する．

（11）　TR で指定された TSS に記録されている CS，IP に制御を移行する．

以上で TR で指定されたタスクが実行され，GDT，LDT で定義されたセグメントディスクリプタによって，メモリ管理，特権保護が実行される．また，GDT，LDT に定義されたコールゲートまたは IDT に定義された割り込みゲート，トラップゲートによって制御移行における特権保護が実現される．

6-5 タスクスイッチ

〔1〕 **タスクスイッチの動作**　80286の実行をあるタスクから別のタスクに切り換えることを**タスクスイッチ**と呼ぶ．タスクスイッチは次のような手順によって実行される．

（1） 現在のレジスタの値をTRで指定される現在のTSSに保存する．

（2） 次のタスクのTSSディスクリプタのセレクタをTRに代入する．これによって，同時にTRキャッシュにTSSディスクリプタの下位5バイトが代入される．

（3） 新しいTSSに定義されたCS，DS，ES，SS以外のすべてのレジスタの値を対応するレジスタに代入する．

（4） TSSのオフセット42から始まる1ワードに記録されているLDTディスクリプタのセレクタをLDTRに代入する．このとき，同時にLDTディスクリプタの下位5バイトがLDTRキャッシュに代入される．

（5） 新しいTSSに定義されたSS，CS，DS，ESのセレクタ値を対応するセグメントレジスタに代入する．

〔2〕 **タスクスイッチ命令**　以上で80286のレジスタ状態は，今まで実行されてきたタスクとまったく関連のない新しいタスクに移る．そして，80286は上述のタスクスイッチを1つの命令によって，または割り込みによって実行することができる．8MHzの80286では，このタスクスイッチを約23μsで実行できる．タスクスイッチは図6・10に示す命令または割り込みによって発生する．割り込みおよびINT命令によって発生するタスクスイッチについては**6-6**において述べる．ここではソフトウェア命令によって発生するタスクスイッチについて述べる．

タスクスイッチを実行する命令といっても，特別な命令が追加されているわけではない．far JMP命令，far CALL命令のオペランドのセレクタが，セグメントディスクリプタとかコールゲートではなく，TSSディスクリプタを指定するとき，一般のJMP命令，CALL命令が実行されるのではなく，タスクスイッチを実行する．また，IRET命令はFLAGのNTが0のときは，第5章で述べたように割り込みからのリターン処理を実行するが，FLAGのNTが1のとき，IRET命令はタスクスイッチを実行する．IRET命令によるタスクスイッチで

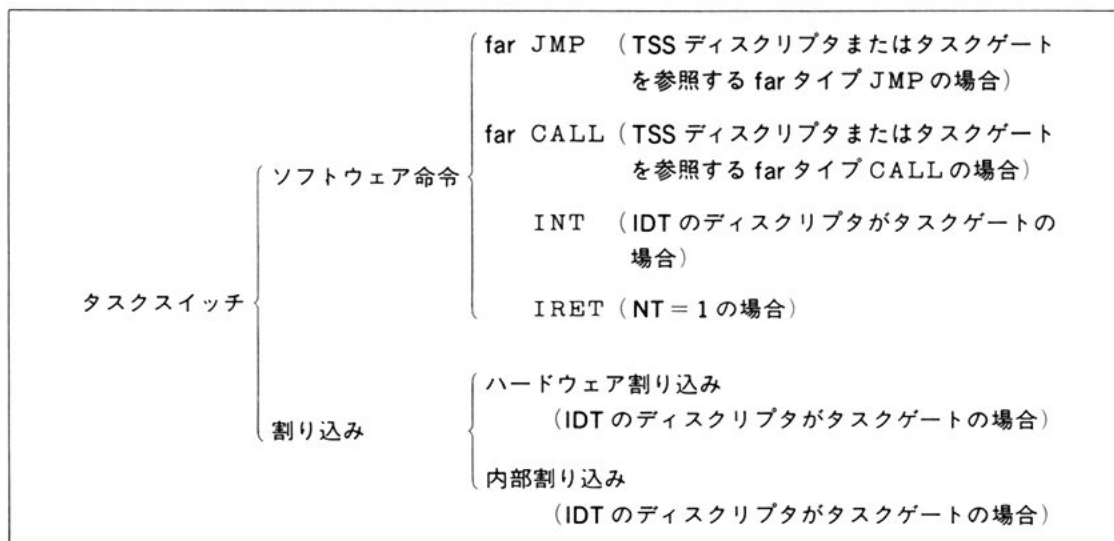



図 6・10 タスクスイッチの発生

は，現在のタスクの TSS のオフセット 0 から始まる 1 ワードに定義されているバックリンクセレクタが，新しいタスクの TSS ディスクリプタのセレクタとなる．このバックリンクセレクタには，CALL 命令または割り込みによってタスクスイッチしたとき，自動的に以前のタスクの TSS ディスクリプタのセレクタが記録されるようになっている．したがって，CALL 命令によってタスクスイッチした後で，IRET 命令によってタスクスイッチすれば，以前のタスクに帰ることができる．

〔3〕 **タスクスイッチ命令の使い分け**　JMP 命令によって実行されても，また CALL 命令，IRET 命令によって実行されても，タスクスイッチの動作自体は変わらない．しかし，命令によって少し異なる部分もある．このことを図 6・11 に示す．これらの命令の違いを考慮して，タスクスイッチの命令を状況に応じて使い分ければよい．図では，タスク A からタスク B にスイッチされ，再びタスク B からタスク A にスイッチされる状況を表している．

JMP 命令によって，タスク A からタスク B にスイッチするとき，タスク A の TSS ディスクリプタの B の状態は，スイッチ前は 0 でも 1 でもかまわないが，スイッチ後は自動的に 0 となる．また，タスク B の TSS ディスクリプタの B の状態は，スイッチ前には必ず 0 でなければならない．そして，スイッチ後は自動的に 1 になる．FLAG の NT もタスクスイッチによって変化し，JMP 命令によるタスクスイッチの場合，0 にクリアされる．JMP 命令によるタスクスイッチ

は最も一般に使用されるものであり，図 6·11 の状況においては，タスク A を終了して (B←0)，タスク B を開始する (B←1) ことを表す．したがって，タスク B の処理において，タスク A にスイッチする JMP 命令を実行する場合も何の問題もなく，タスク A に移ることができる．

CALL 命令によるタスクスイッチの場合，B, NT の変化が JMP 命令で実行される場合とは異なる．CALL 命令の場合，タスク A の TSS ディスクリプタの B の値は変化せず，タスク B の TSS ディスクリプタの B が 0 から 1 になる．そして，FLAG の NT が 1 になる．これは，タスク A の処理は，まだ実行中の

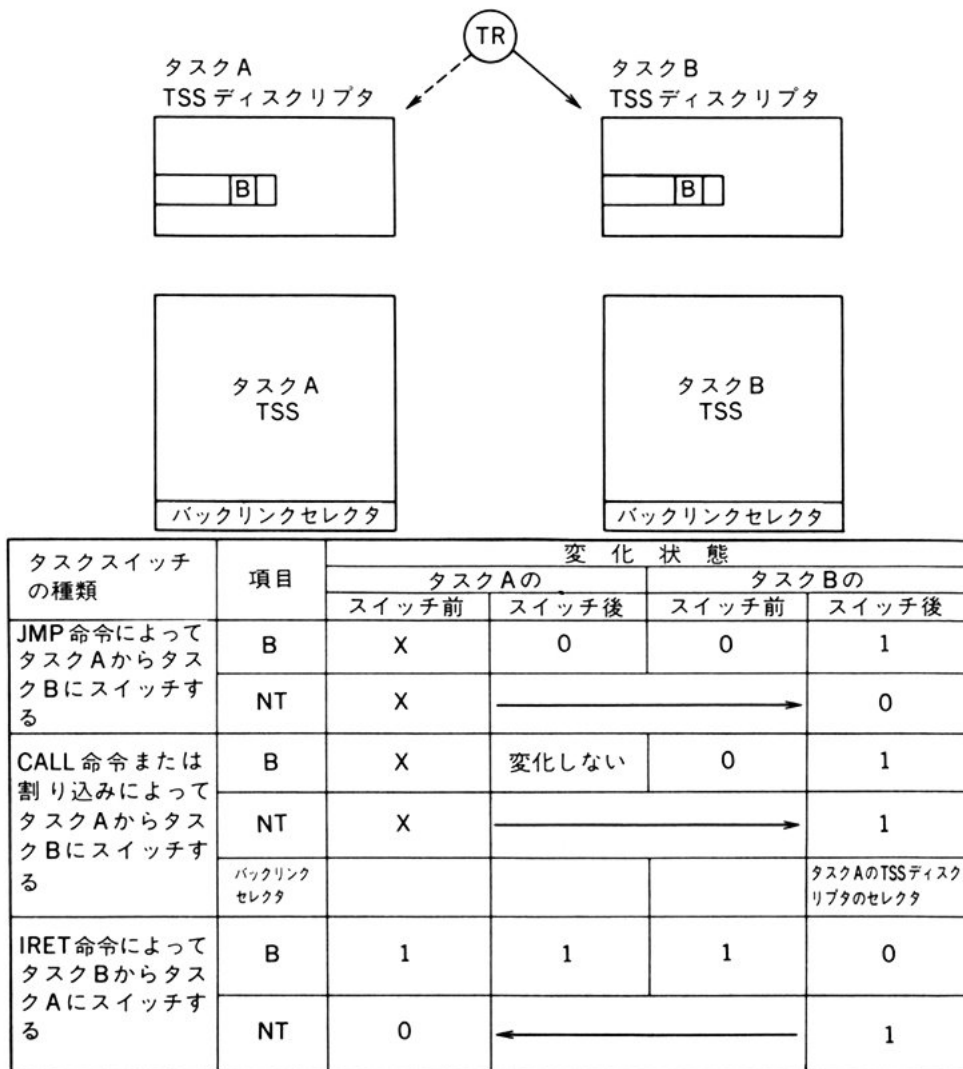
タスクスイッチの種類	項目	変化状態 タスクAの スイッチ前	タスクAの スイッチ後	タスクBの スイッチ前	タスクBの スイッチ後
JMP命令によってタスクAからタスクBにスイッチする	B	X	0	0	1
	NT	X	→		0
CALL命令または割り込みによってタスクAからタスクBにスイッチする	B	X	変化しない	0	1
	NT	X	→		1
	バックリンクセレクタ				タスクAのTSSディスクリプタのセレクタ
IRET命令によってタスクBからタスクAにスイッチする	B	1	1	1	0
	NT	0	←		1

ここで，Xは0でも1でも，どちらでもよいことを表す

図 6·11 タスクスイッチの種類

まま（Bの値が変化しない），新しいタスクBが起動された（B←1）ことを表す．そして，NTが1になるのは，タスクBはタスクAの子プロセスであることを表す．また，タスクBのTSSのバックリンクセレクタには，その親であるタスクAのTSSディスクリプタのセレクタが記録される．このように，CALL命令によるタスクスイッチは，親プロセスが子プロセスを起動するというように考えることができる．

CALL命令によってタスクAからタスクBにスイッチしたとき，タスクBの処理の中でIRET命令を実行すると，CALL命令によるタスクスイッチによってNT＝1となっているから，バックリンクセレクタに定義されたタスクAに戻ることができる．IRET命令によるタスクスイッチは，子プロセスを終了して親プロセスに戻るというイメージで利用することができる．

タスクとプロセス

OSの用語として使用されるタスクという言葉を一言で説明することは困難である．仕事とか処理などの日本語に翻訳してもタスクの意味を説明することにはならない．したがってタスクとは何かと問われたとき，タスクは実行可能なプログラムであると言わざるをえない．実行可能とはプログラムがCPU，メモリなどのコンピュータ資源を利用可能であることを表す．では，1つのタスクは必ず1つのプログラムであるかというとそうではない．タスクの意味を考えるためにはどうしてもマルチタスクシステムを考えなければならない．

マルチタスクシステムは，メモリに配置された実行可能な複数のプログラムがCPU，メモリなどのコンピュータ資源を分け合いながら処理されるシステムである．ここでプログラムが1つのタスクとして処理されるが，もし同種類の2つのタスクを実行する場合には，メモリには1つだけのプログラムを配置し，それを2つのタスクとして2重に使用してもよい．それぞれのタスクが扱う入力データは別々のものであり，また出力するデータも異なる．タスクとはマルチタスクシステムにおいて管理される1処理単位である．

プロセスという言葉も，使用する人によって，さまざまな状況においてさまざまな意味で使われている．しかし，少なくともUNIXのマニュアルの中で使用されているプロセスの意味は80286のタスクと同じである．

6-6 タスクゲート

ゲートと呼ばれるディスクリプタには，コールゲート，割り込みゲート，トラップゲートの他に図6・12に示す**タスクゲート**と呼ばれるゲートがある．タスクゲートには，タスクスイッチをするときの新しいタスクのTSSディスクリプタのセレクタを定義する．タスクゲート自体は，GDT, LDT, IDTに定義することができる．タスクゲートをGDTまたはLDTに定義したとき，far JMP命令，far CALL命令のセレクタ部にタスクゲートのセレクタを指定すれば，タスクゲートに定義したセレクタが指定するTSSディスクリプタのタスクにスイッチできる．このとき，TSSディスクリプタのB, FLAGのNT，バックリンクセレクタの変化は，タスクゲートを参照する命令によって図6・11に示したとおりである．

JMP命令，CALL命令がタスクゲートを参照するとき，タスクゲートのP, DPLについてコールゲートの参照の場合と同様に，P＝1，DPL≧MAX (CPL, RPL) の検査をするので，タスクスイッチの実行を特定の特権レベルに限定することができる．

また，図6・13に示すようにタスクゲートをIDTに定義すれば，割り込みによってタスクスイッチを実行することもできる．割り込みの原因は，INT命令をはじめとするソフトウェア割り込み，ハードウェア割り込み，内部割り込みの任意の原因が可能である．とにかく，割り込みによって参照されたIDTのスロットにタスクゲートが定義されていれば，通常の割り込み処理が実行されるのではなく，タスクスイッチが発生する．したがって，特権レベルの変更によるスタックの切り換え，FLAG，戻りアドレスのCS, IPのPUSHなどは，タスクスイッ

図 6・12　タスクゲート

チではいっさい実行されない．ただし，第7章で述べる保護例外による割り込みでタスクスイッチが実行されるとき，もしエラーコードがあればタスクスイッチの後，新しいタスクのスタックにエラーコードがPUSHされる．

割り込みによるタスクスイッチのとき，TSSディスクリプタのB, FLAGのNT，そして，バックリンクセレクタの変化は，図6･11に示したCALL命令によるタスクスイッチの場合とまったく同じである．したがって，割り込みによってタスクスイッチした後，NTが1になるから，IRET命令を実行すればバックリンクセレクタに記録された以前のタスクに戻ることができる．

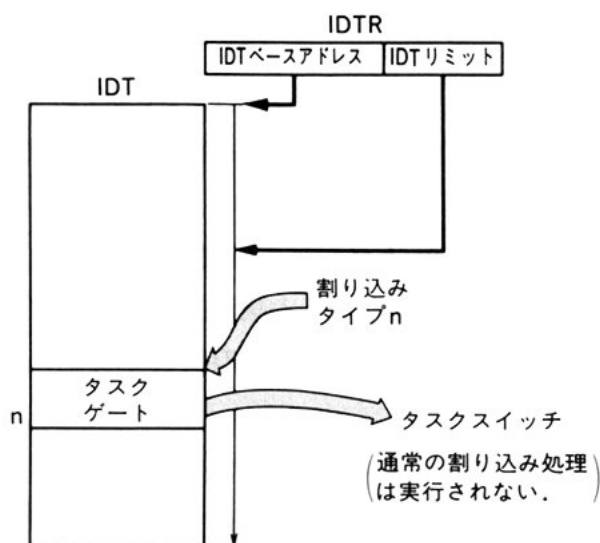



図 6・13　割り込みによるタスクスイッチ

6-7 タスクスイッチの例

OS設計において，タスクスイッチを実行する手続きを一般に**ディスパッチャ**と呼ぶ．ディスパッチャは，いくつかのタスクの中から次に実行するタスクを決定し，そのタスクにスイッチする手続きである．図6·14に示すように，タスクAとタスクBの間で必要に応じてタスクスイッチをしながら処理を進めるシステムを考える．タスクAのプログラムに属するすべてのセグメントは，LDT_Aという LDT によって管理する．また，タスクBのプログラムで定義されたすべてのセグメントは，LDT_Bという LDT によって管理する．手続きディスパッチャは，タスクA，タスクBの両方から共通に参照される手続きであるから，そのセグメントディスクリプタをGDTに定義すればよい．また，一方のタスクの暴走などによって，ディスパッチャコード，データが書き換えられるようなことがあってはならないから，ディスパッチャのセグメントは特権レベル0に定義し，その他のセグメントは特権レベル3に定義することになる．

ディスパッチャを引用する方法は，コールゲートを介したCALL命令を使用することもできるし，また割り込みゲートあるいはトラップゲートをIDTに定義して割り込みによって引用することもできる．外部の周辺装置からデータを入

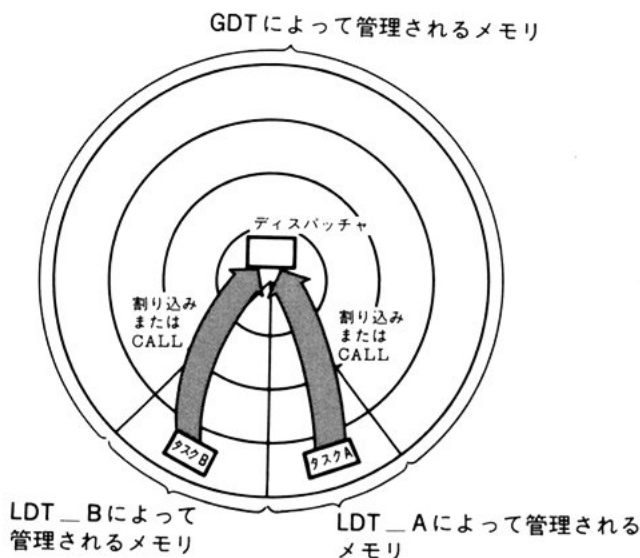



図 6・14 ディスパッチャ

```
NAME dispatcher

PUBLIC dispatcher

dispatch_data SEGMENT RW
       task DD 00180000H,00200000H

       task_ptr DW 0
dispatch_data ENDS

stacka STACKSEG 100H

stackb STACKSEG 100H

code SEGMENT EO
       ASSUME DS:dispatch_data

  dispatcher PROC FAR
             CLI
             PUSH DS
             PUSH AX
             PUSH DX
             PUSH BX

             MOV AX,dispatch_data
             MOV DS,AX

             ADD task_ptr,4        ➡task_ptr に 4 を加算する.
             MOV AX,task_ptr       ➡task_ptr を 8 で割った余りを求め DX に代入する.
             XOR DX,DX
             MOV BX,8
             DIV BX                ➡DX に余りが, そして AX に商が求められる.

             MOV task_ptr,DX
             MOV BX,DX

             JMP task[BX]          ➡タスクスイッチを実行する.
                                   ➡DX＝0 のとき, タスク tsk4d_tss からタスク
                                     exam4d_tss に, また BX＝1 のときタスク
                                     exam4d_tss からタスク tsk4d_tss へ
             POP BX                  スイッチする.
             POP DX
             POP AX
             POP DS
             STI
             IRET
    dispatcher ENDP
 code ENDS
NTAINS PRIVILEGED INSTRUCTIONS
 END
```

図 6・15　ディスパッチャによるタスクスイッチの例

力しなければならないとき，ディスパッチャをCALLして，他のタスクに制御を渡し，必要なデータが入力されたとき再び制御を取り戻すようなシステムを作ることもできるし，また，タイマによって一定時間ごとに割り込みをかけ，ディスパッチャを引用して，タスクを切り換えるようなタイムシェアリングシステムを作ることもできる．

図6·15に非常に単純化したディスパッチャの例を示す．ハードウェアのタイマによって，一定時間ごとにタイプ32の割り込みを発生し，手続きdispatcherに制御を移行するようにする．手続きdispatcherは，配列taskに2つのタスクのTSSディスクリプタのセレクタを初期設定し，dispatcherが引用されるたびに，間接JMP命令すなわちJMP　task［BX］が配列taskの要素を交互に参照して，タスクスイッチを実行する．このために，変数task_ptrを，dispatcherが引用されるたびに，0から4へ，4から0へというように変化させている．このdispatcherによってスイッチされるタスクA，タスクBの例を，図6·16，図6·17に示す．しかし，いまこの2つのプログラムタスクAとタスクBで何をするかということにはあまり興味はない．それらは図6·16および図6·17に示した以

```
 1      NAME task_a
 2
 3      data SEGMENT RW
 4           d1 DW ?
 5           d2 DW ?
 6           rst DW ?
 7      data ends
 8
 9      stack STACKSEG 100H
10
11      code SEGMENT EO
12             ASSUME DS:data
13             ASSUME SS:stack
14
15        start:
16             MOV d1,2
17             MOV d2,4
18             MOV AX,d1
19             ADD AX,d2
20             MOV rst,AX
21             JMP start
22      code ENDS
23      END start,DS:data,SS:stack
```

図6·16 タスクA

```
1       NAME task_b
2
3       data SEGMENT RW
4           buff DB 100H DUP(?)


5       data ENDS
6
7       stack STACKSEG 100H
8
9       code SEGMENT EO
10              ASSUME DS:data
11              ASSUME SS:stack
12
13          start:
14              MOV AX,SEG buff
15              MOV ES,AX
16              MOV DI,OFFSET buff
17              MOV CX,LENGTH buff
18              CLD
19              XOR AL,AL
20          again:
21              STOSB
22              INC AL
23              LOOP again
24
25              JMP start
26      code ENDS
27      END start,DS:data,SS:stack
```

図 6・17 タスク B

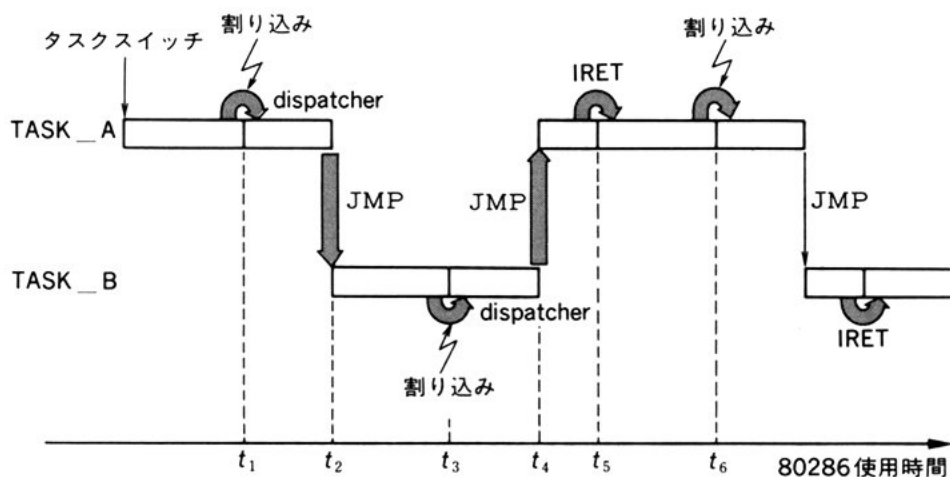



図 6・18 タスクスイッチにおける 80286 処理の流れ

外の任意のプログラムであってもかまわない.

ここでは，このプログラム例で80286がどのように使用されるかについて考える．そのために，80286の実行の時間的な流れを図6·18に示す.

タスクAが最初に実行されるものとする．このような初期タスクもタスクスイッチの命令を使用して起動することができる．しばらくして，タイプ32の割り込みが発生して，タスクAのメインプログラムから手続きdispatcherに制御が移行する．dispatcherの処理の中で次に実行するタスクを決定し，そのタスクに制御を渡す．この場合は，図6·15のJMP task[BX]がタスクスイッチの命令になる．ここで，80286は現在のレジスタの状態をタスクAのTSSに保存し，逆にタスクBのTSSに保存されているレジスタ状態を80286のレジスタに展開してから処理を続ける．このとき，タスクBがプログラムのどこから実行するかということはTSSの初期値によって決まる.

タスクBを実行しているとき，再びタイマからタイプ32の割り込みが入ると，タスクBのメインプログラムからdispatcherに制御が移行する．ここで，タスクAの場合と同じ手続きdispatcherが引用されているわけであるが，今度のdispatcherはタスクBで動作していることに注意するべきである．すなわち，dispatcherはメモリ上に1つしか存在しないにもかかわらず，タスクAで実行されるdispatcherとタスクBで実行されるdispatcherは，その動作においてまったく無関係である.

このタスクBで実行されるdispatcherによって，再びタスクBからタスクAにスイッチされる．このとき，以前，タスクAのTSSに保存された状態が再び80286のレジスタに展開されるから，タスクAの処理は，図6·15に示すdispatcherのタスクスイッチの命令の次の命令POP BXから再開される．そして，dispatcherの最後のIRET命令によって割り込み処理から，再びメインプログラムに戻る．この後は同様の処理が繰り返し実行される.

7. 保護例外

80286の内部割り込みの中で，保護例外と呼ばれているものがある．保護例外による割り込みでは，スタックにFLAG，戻りアドレスのCS，IPをPUSHし，さらに1ワードのエラーコードをPUSHする．保護例外はプロテクトモードにおけるメモリ管理，タスク管理に関係する例外割り込みで，基本的にプロテクトモードにおいてのみ作用する．

7-1 保護例外

保護例外は不正なディスクリプタ，TSSの参照または特権規則に違反した場合に発生する例外割り込みである．保護例外の種類を表7·1に示す．保護の原因によって，割り込みタイプが定義されている．保護例外は，割り込み処理の最初の命令を実行する前に，スタックに図7·1に示すような1ワードの**エラーコード**をPUSHすること以外は一般の割り込みと同じである．

エラーコードのビット0が1のとき，外部からのハードウェア割り込みまたはシングルステップによる割り込みを処理中に，保護例外が発生したことを表す．逆に，ビット0が0のときは，プログラム中の命令が保護例外の原因になっていることを表す．

エラーコードのビット1からビット15までは，保護例外の原因となったディスクリプタが存在するテーブルとインデックスを表す．IDTに定義するディスクリプタ（すなわち，ゲート）の最大のインデックスは255であるから，エラーコードのビット1が1のとき，ビット11からビット15まではすべて0である．また，保護例外の原因によっては，エラーコードはすべて0の場合もある．

保護例外に対応するIDTのディスクリプタには，割り込みゲート，トラップゲートまたはタスクゲートを定義すればよい．割り込みゲートまたはトラップ

表7·1　保護例外の種類

割り込みタイプ	保護例外の種類	エラーコード	割り込みゲートまたはトラップゲートを使用したときスタックにPUSHされる戻りアドレス	リアルモード	プロテクトモード
8	2重エラー	0	保護例外の原因となった命令の先頭アドレス	特定条件で作用する	作用する
10	不正TSS	原因によって決まる		作用しない	作用する
11	セグメント不在	原因によって決まる		作用しない	作用する
12	スタックエラー	原因によって決まる		作用しない	作用する
13	一般保護エラー	原因によって決まる		特定条件で作用する	作用する

ゲートを使用するとき，割り込み手続きによって保護例外を処理することができる．また，タスクゲートを使用するときは保護例外を別のタスクで処理することができる．

2	1	ディスクリプタテーブル
X	1	IDT
0	0	GDT
0	1	LDT

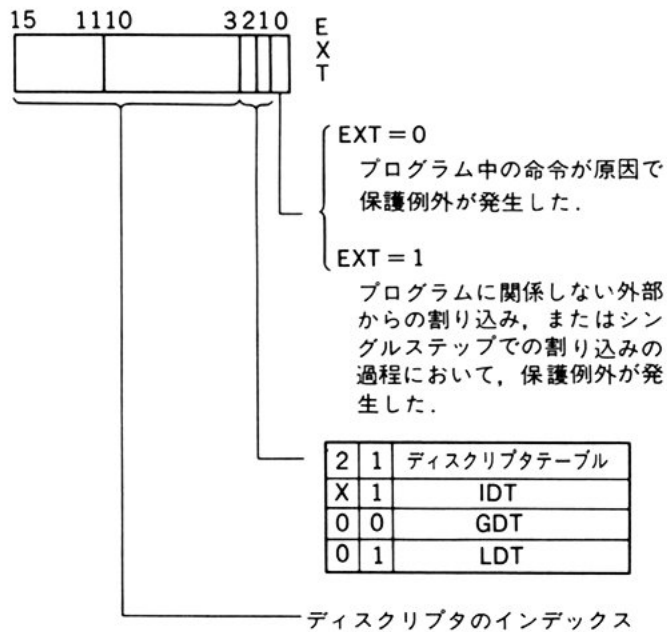



図 7・1　保護例外のエラーコード

7-2 スタックエラー

スタックエラーは，SSレジスタを使用したメモリ参照において，セグメントリミットを超えた場合，またはSSに代入するセレクタがP＝0のディスクリプタを指定しているときに発生し，タイプ12の割り込み処理を実行する．

PUSH命令，POP命令，CALL命令，割り込みなどの他に，MOV AX，[BP]のような命令によっても，前者の原因でスタックエラーが発生する．このとき，タイプ12の割り込み処理を実行する前に，0のエラーコードがPUSHされる．

後者の原因でのスタックエラーは，SSに新しいセレクタを代入するすべての命令において発生する可能性がある．もちろん，タスクスイッチの命令の中でも，SSに新しいセレクタを代入する処理が含まれるので，スタックエラーが発生する可能性はある．

タスクスイッチ中にスタックエラーが発生した場合，付録Bのタスクスイッチのシーケンスからわかるように，タイプ12の割り込み処理は新しいタスクでで実行される．しかし，タスクスイッチの際に発生するスタックエラーはDSキャッシュまたはESキャッシュにディスクリプタを代入する前に発生し，DSキャッシュ，ESキャッチュは未定義のまま残ってしまう．したがって，割り込み手続きの先頭で

```
MOV  AX,DS
MOV  DS,AX
MOV  AX,ES
MOV  ES,AX
```

のような処理を実行して，DSおよびESのキャッシュを定義する必要がある．あるいは，タイプ12の割り込みにタスクゲートを使用して，スタックエラーを別のタスクで処理してもよい．

特権レベル0において発生したスタックエラーは，必ず別のタスクで処理するべきである．そうでなければ，スタックエラーの割り込みにおいて，再び同じスタックセグメントを使用することになるからである．

7-3 TSS エラー

TSS エラーは，タスクスイッチ実行中に表 7·2 に示す状況において発生し，タイプ 10 の割り込みを実行する．TSS エラーのエラーコードのインデックス部はその原因によって，表に示すように決まる．

IDT のタイプ 10 のゲートには，必ずタスクゲートを定義し，TSS エラーを別のタスクで処理しなければならない．

表 7·2　TSS エラーを発生する原因

原　　因	エラーコード
TSS ディスクリプタのリミット < 43	TSS ディスクリプタのインデックス
LDT のセレクタが不正なディスクリプタを指定する．または LDT ディスクリプタの P が 0 である．	LDT ディスクリプタのインデックス
SS のセレクタがテーブルリミットを超えている．	SS ディスクリプタのインデックス
SS のディスクリプタにおいて，W＝0	SS ディスクリプタのインデックス
SS のディスクリプタにおいて，DPL ≠ CPL	SS ディスクリプタのインデックス
SS のセレクタにおいて，RPL ≠ CPL	SS ディスクリプタのインデックス
CS のセレクタがテーブルリミットを超えている．	CS ディスクリプタのインデックス
CS のディスクリプタにおいて，E＝0	CS ディスクリプタのインデックス
CS のディスクリプタにおいて，C＝0 のとき，DPL ≠ CPL	CS ディスクリプタのインデックス
CS のディスクリプタにおいて，C＝1 のとき，DPL > CPL	CS ディスクリプタのインデックス
DS，ES のセレクタがテーブルリミットを超えている．	DS または ES ディスクリプタのインデックス
DS，ES のディスクリプタにおいて，E＝1 かつ R＝0	DS または ES ディスクリプタのインデックス

この表において，LDT のセレクタ，SS のセレクタ，CS のセレクタ，DS のセレクタ，ES のセレクタ，CPL，RPL はすべてタスクスイッチにおいて，新しいタスクの TSS に保存されている値を表す．

7-4 Pビットエラー

Pビットエラーは，P＝0のディスクリプタを，セグメントキャッシュ，TRキャッシュ，LDTRキャッシュに代入するときに発生し，タイプ11の割り込み処理を実行する．このとき，エラーコードは，原因となったディスクリプタのセレクタを表す．

3-5で述べたように，Pビットエラーを利用すれば，1タスク当たり1Gバイトの仮想メモリ空間を実現することができる．しかし，仮想メモリはハードディスクなどの補助記憶装置と実メモリとの間でスワップを実行するため，大きなメモリ空間を必要とするマルチユーザのタイムシェアリングシステムにおいては有効であるが，タイミングの重要な工業用のリアルタイム制御には向かない．この場合，すべてのディスクリプタのPを1にしておけば，Pビットエラーの保護例外は発生しない．

タスクスイッチ中に発生するPビットエラーを，割り込みゲートまたはトラップゲートを使用して処理する場合にも，80286のタスクスイッチ処理の順序から，DS，ESのセレクタは定義されているが，DSおよびESのキャッシュにはディスクリプタが定義されていない．したがって，このような状態のPビットエラーをタスクゲートを使用して，別タスクで処理するならば問題はないが，割り込みゲートまたはトラップゲートを使用して，割り込み手続きで処理するときには

```
MOV  AX,DS
MOV  DS,AX
MOV  AX,ES
MOV  ES,AX
```

の命令を実行して，DSおよびESのキャッシュを定義しなければならない．

7-5 一般保護エラー

一般保護エラーは，スタックエラー，TSSエラー，Pビットエラーそして次に述べる2重エラー以外の一般的な保護例外によって発生し，タイプ13の割り込み処理を実行する．一般保護エラーの原因としては，たとえばセグメントリミットを超えたメモリ参照，E＝1のセグメントへのデータライト，W＝0のセグメントに対するデータリード，ヌルセレクタ(0)をもつDS，ESを使用したメモリ参照，特権レベルによる保護，不正なディスクリプタの参照などがある．

一般保護エラーのエラーコードは，状況によって0またはエラーの原因となったディスクリプタのセレクタが，図7·1に示した形式でスタックにPUSHされる．

保護例外の効用

8086のアセンブリ言語のプログラミングで最初よく犯すうっかりミスは，DS，ESなどのセグメントレジスタの初期設定を忘れることである．プログラミングに習熟してくれば，「プログラムの実行開始時には必ずセグメントレジスタの初期設定をする」また「farタイプの手続きの先頭では，セグメントレジスタをスタックにPUSHしてから新しい値を代入する」というようにプログラムのスタイルが決まってくるので，上記のようなミスを犯すことはまずない．しかし，何かの都合によってプログラムのスタイルが変わるようなとき，セグメントレジスタの初期設定を忘れることがある．

8086でやっかいなのは，DSにでたらめな値が入っていてもそれなりに動作してしまうことである．もちろん，まったく別のメモリ空間をデータセグメントとして使用するわけであるから，他のデータを壊すなどして正しい動作はしない．たいていの場合そのプログラムは暴走する．

しかし，プロテクトモードの80286では，DSに誤ったデータが入った場合，誤って動作することはまずない．誤った処理を実行する命令は保護によって例外割り込みが発生する．8086のOSを80286のプロテクトモードに移植したとき，それまで発見されなかったバグが発見できた例もある．

7-6 2重エラー

2重エラーは，1つの命令において2重の保護が働いたときに発生し，タイプ8の割り込みを実行する．たとえば，一般保護エラーが発生して，その割り込み処理においてスタックにFLAG，戻りアドレスのCS，IPそしてエラーコードをPUSHしたところが，スタックのリミットを超えたような場合に，2重エラーが発生する．

2重エラーに対する処理は，必ず別タスクで実行するべきである．さもなければ3重のエラーが発生する可能性がある．しかし，80286には，3重エラーに対する保護例外はもっていない．もし，2重エラーによるタイプ8の割り込みの過程で再び保護例外が発生すれば，80286は**シャットダウン状態**になり，すべての機能を停止する．シャットダウン状態は，80286がHLT命令を実行した後の**ホルト状態**と同じである．ただし，シャットダウン状態とホルト状態は，アドレスバスのA_1端子のレベルによって，ハードウェア的に見分けることができる．シャットダウン状態のときはA_1＝Lowであり，ホルト状態のときはA_1＝Highである．

また，シャットダウン状態から抜け出すためには，NMI端子に割り込み信号を入れる方法と，80286をリセットする方法がある．

7-7 例外処理と再実行

保護例外は，エラーコードをPUSHする以外は，通常の割り込みと同様である．保護例外に使用するゲートは，例外割り込みが発生する状況によって割り込みゲート，トラップゲートそしてタスクゲートのどれかを使用することができる．図7·2(a)，(b)，(c)に割り込みゲートまたはトラップゲートを使用して，同じ特権レベルの割り込み処理に移行する場合，より高い特権レベルの割り込み処理に移行する場合，タスクゲートを使用して，別のタスクにタスクスイッチした場合のそれぞれのスタックの様子を示す．

割り込みゲートまたはトラップゲートを使用したとき，スタックにPUSHされる戻りアドレスは，保護例外の原因となった命令の先頭アドレスである．また，

(a) 割り込みゲートまたはトラップゲートを使用し，特権レベルが変化しない場合

(b) 割り込みゲートまたはトラップゲートを使用し，特権レベルが変化する場合

(c) タスクゲートを使用する場合

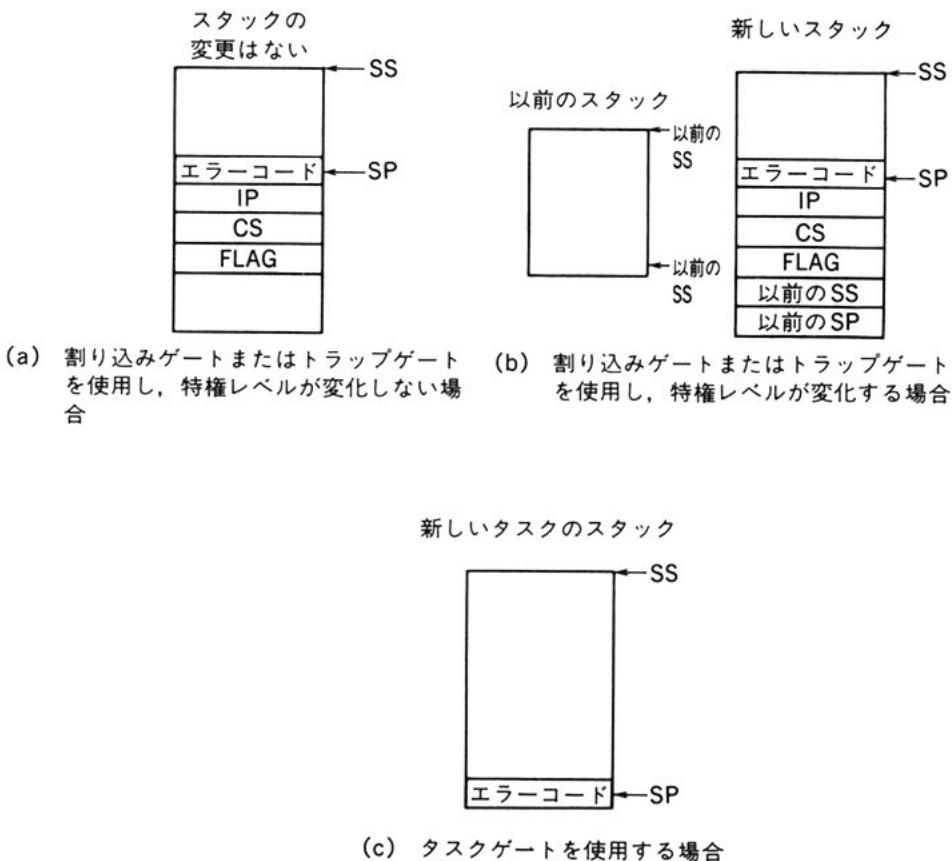



図7·2 保護例外によって，スタックにPUSHされるエラーコード

タスクゲートを使用した場合，TSSのバックリンクセレクタで指定される以前のタスクのTSSには，保護例外の原因となった命令の先頭アドレスを表すCSおよびIPが保存される．したがって，保護例外の原因を取り除いてから

```
ADD SP,2
IRET
```

を実行することによって，保護例外を発生した命令を再実行することができる．

しかし，次の命令は例外的に再実行できない．

（1） セグメントリミットを超えることによる一般保護エラーを発生したREPプリフィックス付きのストリング命令．

（2） ディスクリプタのWが0（リードのみ）のセグメントにライトした，ADC命令，SBB命令，RCL命令，RCR命令．

（3） ディスクリプタのWが0（リードのみ）のセグメントにライトしたXCHG命令．

(1)の場合，ストリング命令でカウンタとして使用するCXの更新をしないまま例外割り込みに移行するため，IRET命令を実行しても繰り返し処理の途中から実行を再開することにならない．(2)の場合は，保護例外が発生するのは演算が終了してからであるから，CFの値は変化してしまう．したがって，IRET命令によって再実行しても，最初の実行と同じ環境では実行できないからである．(3)のXCHG命令が再実行できない理由も，1度目の実行によって他方のオペランドに指定したレジスタの内容が変化してしまうからである．

8. 80286のハードウェア

80286のハードウェア構成は，8086に比較して決して複雑ではない．しかし，80286のバスは，8086のようにアドレスバスとデータバスが共通の端子を使用することはなく，より効率のよいバスサイクルを実現している．また，バスサイクルは2プロセッサクロックで実行され，8086のバスサイクル速度と比較して2倍になっている．CPUのクロックが高速になるとき問題になるのがメモリ，I/Oの応答速度である．CPUのクロックを速くしても，参照するメモリの応答速度が遅いためにウェイトステートクロックを挿入していたのでは意味がない．では，応答速度の速いメモリを使用すればよいが，高価である．また，80286をプロテクトモードで使用するようなシステムでは，1～2Mバイト以上の大容量のメモリを使用することが多い．高速で80286を効率よく動作させるシステムを作るために高価なメモリを大量に使用することは経済的に問題がある．そこで，80286では，比較的安価なメモリを使用しても十分なスピードのシステムを作れるようになっている．

8-1　CPUモジュールの構成

〔1〕 **電力供給とシステムクロック**　80286の端子配置図は図1·5に示した．表8·1，表8·2に各端子の信号と働きを示す．80286への電力供給は，図8·1に示すように，V_{CC}に＋5Vの電圧を与え，V_{SS}を信号グランドに接続する．また，CAP端子と信号グランド間には，0.047μF±20%12Vのコンデンサを接続する．これは80286の内部サブストレートバイアスのフィルタとなる．V_{CC}，V_{SS}端子はそれぞれ複数あり，内部では共通であるが，これらはすべて接続するようにする．

CLK端子にはシステムクロックを供給する．80286の内部回路は，このシステムクロックを2分周したプロセッサクロックに同期して動作する．すなわち，8MHzで動作する80286には，16MHzのシステムクロックを供給する．このシステムクロックは，図8·1に示すように，**クロックジェネレータ82284**によって作られる．

表8·1　80286端子一覧表

信号と記号	信号の方向	説　明
CLK（システムクロック）	入力	システムクロックを入力する．80286内部の分周回路によってCLKを2分周して，プロセッサクロック（PCLK）が作られる．80286内部の各回路はPCLKに同期して動作する．
D_0-D_{15}（データバス）	入出力	メモリまたはI/Oインタフェースとの間でデータを入出力するために使用する．
A_0-A_{23}（アドレスバス）	出力	メモリまたはI/Oインタフェースにアドレスを出力するために使用する．
$\overline{BHE}$（バスハイイネーブル）	出力	データの入出力において，データバスの上位バイト（D_8-D_{15}）を使用するときLowが出力され，奇数アドレスのメモリ，I/Oに対してデータの入出力が行われることを表す．
$\overline{S_0}$　$\overline{S_1}$（バスサイクルステータス）	出力	COD/$\overline{INTA}$，M/$\overline{IO}$と組み合わせ，80286のバスサイクルの状態が表8·2のように表される．
M/$\overline{IO}$（メモリ，I/O）	出力	メモリに対するバスサイクルのときHighを出力し，I/Oに対するバスサイクルのときLowを出力する．
COD/$\overline{INTA}$（コード，割り込みアクノリッジ）	出力	M/$\overline{IO}$＝Highの状態においてCOD/$\overline{INTA}$＝Highはコードフェッチを表す．またM/$\overline{IO}$＝Lowの状態においてCOD/$\overline{INTA}$＝Lowは割り込みアクノリッジを表す．

信号と記号	信号の方向	説　　明
$\overline{\text{LOCK}}$ (バスロック)	出力	LOCK 付き命令，XCHG 命令，割り込みアクノリッジ，ディスクリプタテーブル参照によって実行されるメモリ参照のバスサイクルにおいて Low を出力する．システムバス制御に利用する．
$\overline{\text{READY}}$ (バスレディ)	入力	バスサイクル終了のタイミングを制御する．T_C の最後で $\overline{\text{READY}}$ が Low のときバスサイクルは終了し，$\overline{\text{READY}}$ が High のとき再び T_C を繰り返す．
HOLD (バスホールドリクエスト)	入力	ローカルバスの所有権を制御する．他のハードウェアが 80286 の HOLD を High にすることによって 80286 はバスの各端子を OFF 状態にし，その確認信号として HLDA を High にする．HLDA が High になっている間，80286 以外のハードウェアがローカルバスを制御することができる．
HLDA (ホールドアクノリッジ)	出力	
INTR (割り込みリクエスト)	入力	割り込みコントローラ 8259A からの割り込み要求信号を入力する．割り込み要求信号が 80286 内部でマスクされていなければ割り込みアクノリッジサイクルを実行し，8259A から割り込みタイプをリードし，割り込みを発生する．
NMI (ノンマスカブル割り込みリクエスト)	入力	エッジトリガの信号を入力することによって，80286 は無条件にタイプ 2 の割り込みを発生する．
PEREQ (80287 リクエスト)	入力	80286 はメモリ参照を伴う ESC 命令を実行したとき，数値演算プロセッサ 80287 が出力する PEREQ 信号を入力可能にする．実際に 80287 から PEREQ 端子に High の信号が送られたとき，80286 は 80287 が要求するバスサイクルを実行する．また，その最初のバスサイクルにおいて，80287 へのアクノリッジ信号として $\overline{\text{PEACK}}$ に Low のパルスを出力する．
$\overline{\text{PEACK}}$ (80287 アクノリッジ)	出力	
$\overline{\text{BUSY}}$ (80287 ビジー)	入力	数値演算コプロセサ 80287 の動作状態を 80286 に知らせる．$\overline{\text{BUSY}}$ 端子が Low になっている状態では再び $\overline{\text{BUSY}}$ 端子が High になるまで 80286 は ESC 命令，WAIT 命令を実行しない．
$\overline{\text{ERROR}}$ (80287 エラー)	入力	教値演算コプロセサ 80287 から 80286 に例外処理を要求するために使用する．80286 においてタイプ 16 の割り込みを発生する．
RESET (システムリセット)	入力	80286 の内部状態を初期化する．
V_{SS} (信号グラウンド)	入力	電圧 0 [V] に接続する．
V_{CC} (電源)	入力	＋5 [V] の電源を供給する．
CAP (サブストレートフィルタキャパシタ)	入力	CAP には V_{SS} との間に 0.047 μF ± 20% 12 V のコンデンサを接続する．これは 80286 内部のサブストレートバイアスジェネレータ出力のフィルタとなる．

82284 の端子配置図とブロック図を図 8·2 に示す．X_1 と X_2 の間に水晶振動子を接続することによって，内部のクリスタルオシレータが発振し，CLK 端子にクロックを出力する．また，内部のクリスタルオシレータを使用せず，外部のオシレータの出力を EFI 端子に接続し，クロックを得ることができる．内部のオシレータを使用するか，EFI 端子からの入力信号を使用するかは，F/$\overline{C}$ によって決める．F/$\overline{C}$ を Low にしたとき，内部のオシレータの出力がクロックとして CLK 端子に出力され，F/$\overline{C}$ を High にしたとき，EFI 端子の入力信号がクロックとして CLK 端子に出力される．また，82284 の PCLK 端子には CLK 端子に出力されるシステムクロックを 2 分周したクロックが出力されている．この PCLK は，80286 のプロセッサクロックに同期し，82284 の PCLK が High のとき CLK の立ち下がりがプロセッサクロックの ϕ1 の終了のタイミングになる．

〔2〕 **RESET と $\overline{\text{READY}}$**　82284 はシステムクロックを 80286 に供給する

表 8·2 ステータス信号とバスサイクル

COD/$\overline{\text{INTA}}$	M/$\overline{\text{IO}}$	$\overline{S_1}$	$\overline{S_0}$	バスサイクル
0	0	0	0	割り込みアクノリッジ
0	0	0	1	この組み合わせの信号は発生しない．
0	0	1	0	この組み合わせの信号は発生しない．
0	0	1	1	アイドル状態
0	1	0	0	A1＝1 の場合　ホルト状態 A1＝0 の場合　シャットダウン状態
0	1	0	1	メモリデータリード
0	1	1	0	メモリデータライト
0	1	1	1	アイドル状態
1	0	0	0	この組み合わせの信号は発生しない．
1	0	0	1	I/O リード
1	0	1	0	I/O ライト
1	0	1	1	アイドル状態
1	1	0	0	この組み合わせの信号は発生しない．
1	1	0	1	コードフェッチ
1	1	1	0	この組み合わせの信号は発生しない．
1	1	1	1	アイドル状態

だけではなく，システムクロックに同期した $\overline{\textbf{READY}}$ 信号と **RESET** 信号を供給する．RESET 信号は図 8・3 に示すように，16 CLK 以上の長さをもつパルスを 80286 に入力して，80286 の内部状態を初期化する．80286 に電源を投入するときには，必ず RESET 信号をも供給しなければならない．ただし，電源投入時の RESET 信号は 50 ms 以上の間 High にしてから，Low に下がるようなパルスにする．これは，CAP 端子に接続するコンデンサの充電時間である．

RESET 信号が Low になるとき，80286 の信号出力端子は図 8・3 に示すように High または Low に決まり，RESET 信号が Low になってから，約 38 CLK 後に最初のバスサイクルが実行される．このとき 80286 の各レジスタの値は表 8・3 に示すように決まる．したがって，リセット後の 80286 はリアルモードで始まり，物理アドレスが 0FFFFF0H 番地のメモリから最初の命令をフェッチして，実行する．このアドレスには，一般にシステムの初期設定を実行するプログラムに制御を移行する JMP 命令を定義する．

80286 に電源を供給したとき確実にリセットするために，82284 の $\overline{\text{RES}}$ 端子に図 8・1 に示したような RC 回路を接続する．$\overline{\text{RES}}$ 端子の入力電圧は，82284 内部のシュミットトリガ回路によって波形整形された後，CLK の立ち下がりに同期して RESET 端子に出力される．

82284 から 80286 に供給される $\overline{\text{READY}}$ 信号によって，80286 のバスサイクル

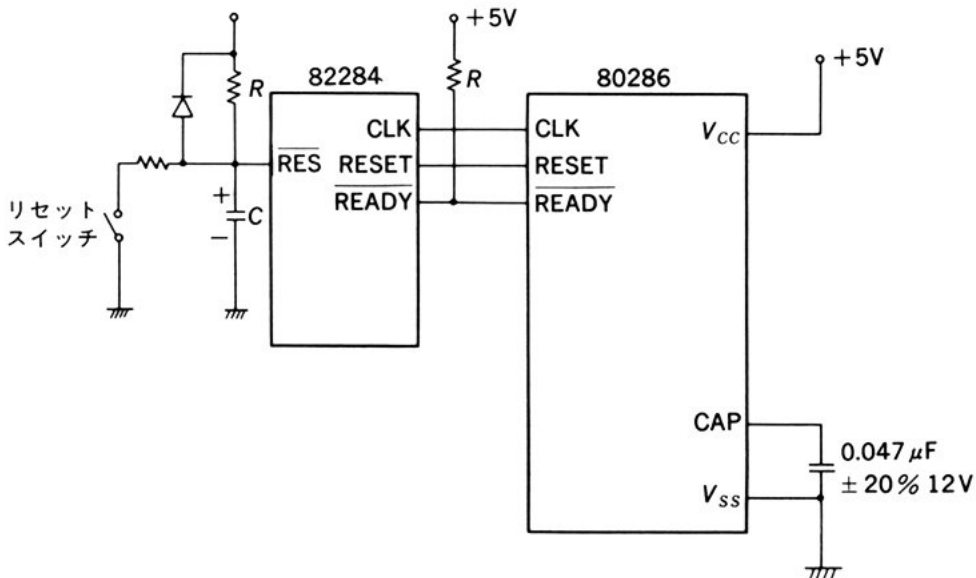



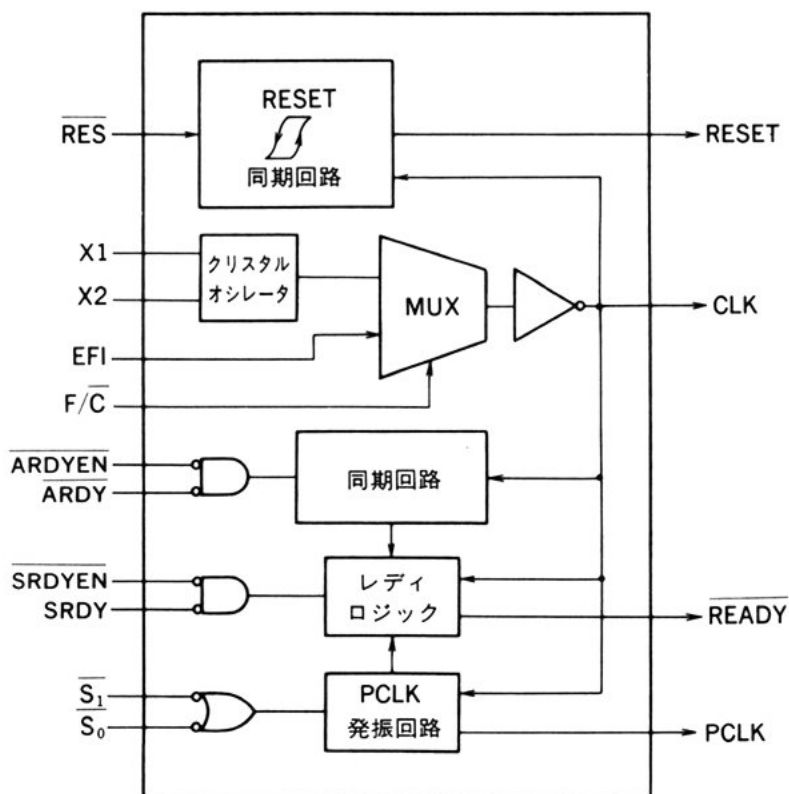
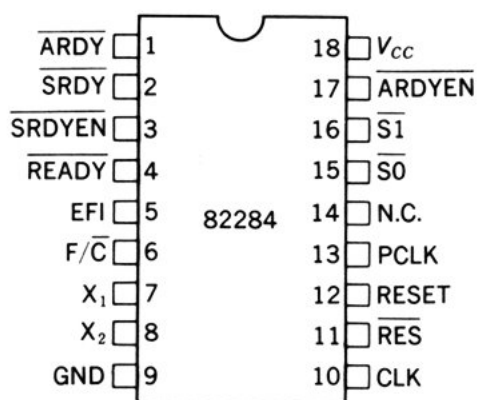
図 8・1　電力供給と 82284 の接続

(a) 82284ブロック図

(b) 82284ピン配置図

図 8・2 82284クロックジェネレータ
(iAPX 286 Hardware Reference Manual (Order No. 210760-001), A-75, Fig 1, Fig 2より引用)

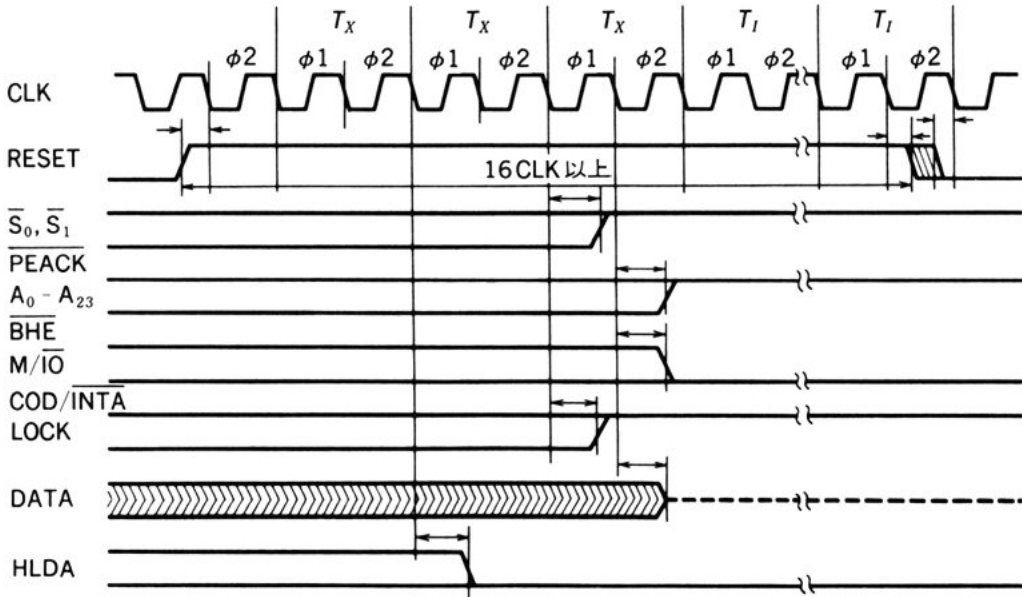



図 8・3 リセット信号
(iAPX 286 Hardware Reference Manual (Order No.210760-001), 3-68, Fig 3-69より引用)

表 8・3 80286リセット後のレジスタの状態

レジスタ名	初 期 値
FLAG	0002H
MSW	0FFF0H
IP	0FFF0H
CS	0F000H
DS	0000H
SS	0000H
ES	0000H
CS キャッシュ	ベースアドレス＝0FF0000H, リミット＝0FFFFH
DS キャッシュ	ベースアドレス＝000000H, リミット＝0FFFFH
SS キャッシュ	ベースアドレス＝000000H, リミット＝0FFFFH
ES キャッシュ	ベースアドレス＝000000H, リミット＝0FFFFH
IDTR	ベースアドレス＝000000H, リミット＝03FFH

の長さが決まる．図8·4に示すように，80286のバスサイクルは，T_s，T_cの2つのプロセッサクロックによって構成されるが，T_cの最後に$\overline{\text{READY}}$信号をHighにすることによって，T_cの後にもう1つのT_cを挿入し，バスサイクルを

延長することができる．T_cの最後に$\overline{\text{READY}}$信号をLowにしたとき，バスサイクルは終了する．

82284の$\overline{\text{READY}}$端子はオープンコレクタであるから，図8·1に示したようにプルアップ抵抗を接続しなければならない．また，このため80286の$\overline{\text{READY}}$端子には，82284の$\overline{\text{READY}}$以外の他のレディ信号をワイヤードオアで接続することが可能である．82284は$\overline{\text{SRDY}}$，$\overline{\text{ARDY}}$の2つの入力端子の信号から，CLKに同期した$\overline{\text{READY}}$を作る．$\overline{\text{SRDY}}$，$\overline{\text{ARDY}}$はそれぞれ，イネーブル端子として$\overline{\text{SRDYEN}}$，$\overline{\text{ARDYEN}}$の2つの端子をもつ．これらのイネーブル端子子をLowにしたときにのみ，図8·2(a)のブロック図からわかるように，$\overline{\text{SRDY}}$，$\overline{\text{ARDY}}$端子からの入力信号がそれぞれ有効に働く．

図8·5に$\overline{\text{SRDY}}$端子からの入力信号と$\overline{\text{READY}}$端子の出力信号の関係を示す．CLKの立ち下がりにおいて，82284に入力される$\overline{S_0}$，$\overline{S_1}$の少なくとも一方がLowであれば，82284は$\overline{\text{READY}}$端子をハイインピーダンスの状態にする．82284の$\overline{\text{READY}}$端子はプルアップされているから，バスサイクルの最初に$\overline{\text{READY}}$を一度Highにすることができる．この後，$\overline{S_0}$，$\overline{S_1}$の両方がHighになっている状態で，PCLKがHighの状態におけるCLKの立ち下がりにおいて，82284は$\overline{\text{SRDYEN}}+\overline{\text{SRDY}}$の入力信号を調べ，$\overline{\text{SRDYEN}}+\overline{\text{SRDY}}$がLowであれば，$\overline{\text{READY}}$端子にLowを出力する．

また図8·6に$\overline{\text{ARDY}}$の入力信号と$\overline{\text{READY}}$の出力の関係を示す．82284はCLKの立ち下がりにおいて，$\overline{\text{ARDYEN}}+\overline{\text{ARDY}}$の状態を調べ，それがLowであれば，次のPCLKがHighの状態におけるCLKの立ち下がりにおいて$\overline{\text{READY}}$をLowにする．したがって，$\overline{\text{ARDY}}$を使用するとき，$\overline{\text{READY}}$は最少でも1CLKだけLowになるタイミングが遅れるが，プロセッサクロックの$\phi 1$の立ち下がりに内部で同期して出力されるため，$\overline{\text{ARDY}}$に入力するレディー信号は非同期でもかまわない．$\overline{\text{SRDY}}$，$\overline{\text{ARDY}}$のどちらかにレディー信号が供給されることによって，$\overline{\text{READY}}$にLowの信号を出力する．

〔3〕 **コマンド信号の生成**　80286がバスサイクルを実行するとき，メモリ，I/Oインタフェースは対応する動作を実行する．このために，メモリまたはI/Oインタフェースにコマンド信号と呼ぶ信号を供給する．しかし，80286はコマンド信号を直接に出力せず，代わりにステータス信号$\overline{S_0}$，$\overline{S_1}$とM/$\overline{\text{IO}}$信号を出力する．80286は$\overline{S_0}$，$\overline{S_1}$，M/$\overline{\text{IO}}$の3つの信号の組み合わせによって，表8·4に示

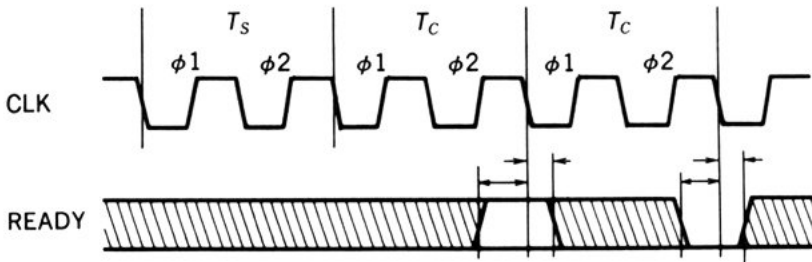



図 8・4 バスサイクルの終了

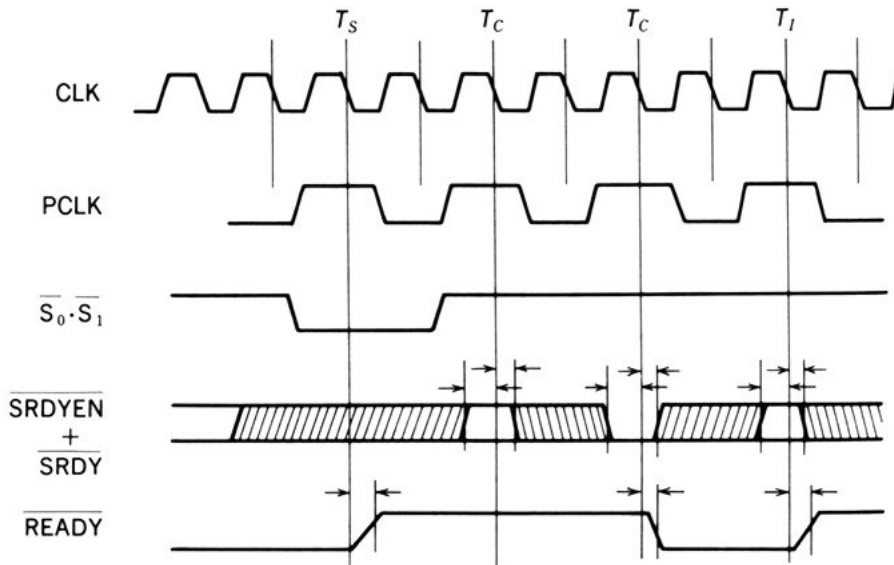



図 8・5 $\overline{\text{SRDY}}$ 制御による 1 ウェイトステートの挿入

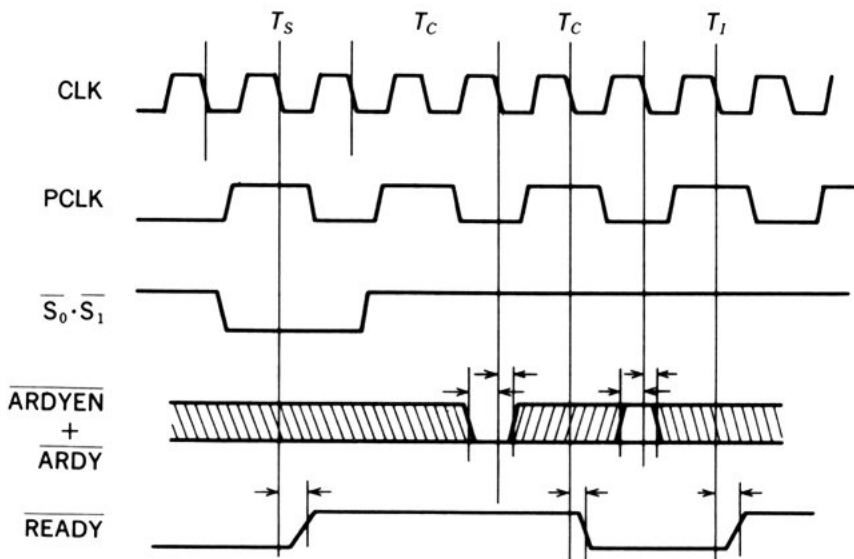



図 8・6 $\overline{\text{ARDY}}$ 制御による 1 ウェイトステート

表 8・4　バスサイクルの種類

$M/\overline{IO}$	$\overline{S_1}$	$\overline{S_0}$	バス動作
0	0	0	割り込みアクノリッジ
0	0	1	I/O リード
0	1	0	I/O ライト
0	1	1	アイドル状態
1	0	0	ホルト（A_1＝1 の場合） またはシャットダウン（A_1＝0 の場合）
1	0	1	メモリリード
1	1	0	メモリライト
1	1	1	アイドル状態

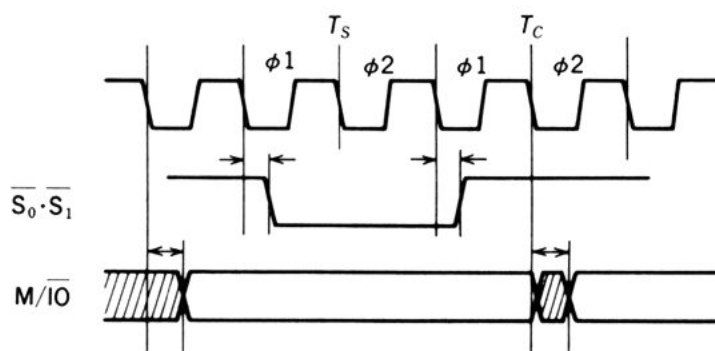



図 8・7　ステータス信号の出力タイミング

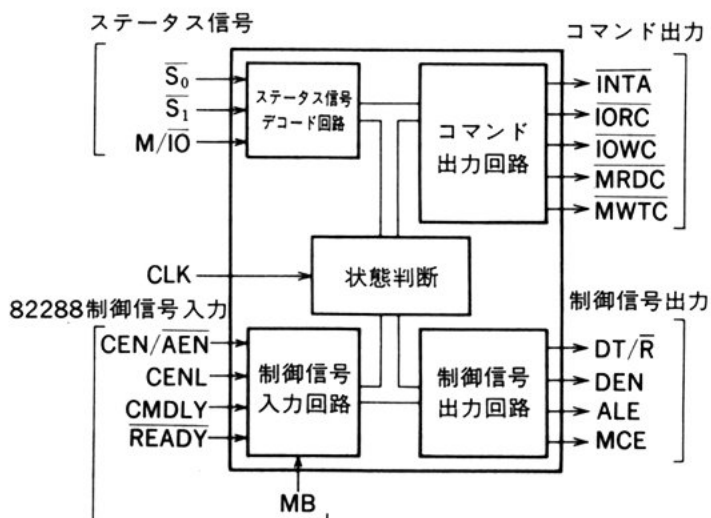



図 8・8　82288バスコントローラ
（iAPX 286 Hardware Reference Manual
（Order No. 210760-001），A-83，Fig 1 より引用）

すように実行するバスサイクルの種類を表現する．また，$\overline{S_0}$，$\overline{S_1}$，M/$\overline{IO}$ が出力されるタイミングを図 8·7 に示す．

しかし，このステータス信号，M/$\overline{IO}$ 信号を直接にメモリ，I/O インタフェースに供給することはできない．ステータス信号と M/$\overline{IO}$ 信号からコマンド信号を作るために，図 8·8 に示す**バスコントローラ 82288** を使用する．82288 は $\overline{S_0}$，$\overline{S_1}$，M/$\overline{IO}$ の信号をデコードして，割り込みアクノリッジ信号 $\overline{INTA}$，I/O リードコマンド $\overline{IORC}$，I/O ライトコマンド $\overline{IOWC}$，メモリリードコマンド $\overline{MRDC}$，メモリライトコマンド $\overline{MWTC}$ を出力する．これらのコマンド信号の出力タイミングは CMDLY 信号によって，またコマンド信号の終了タイミングは $\overline{READY}$

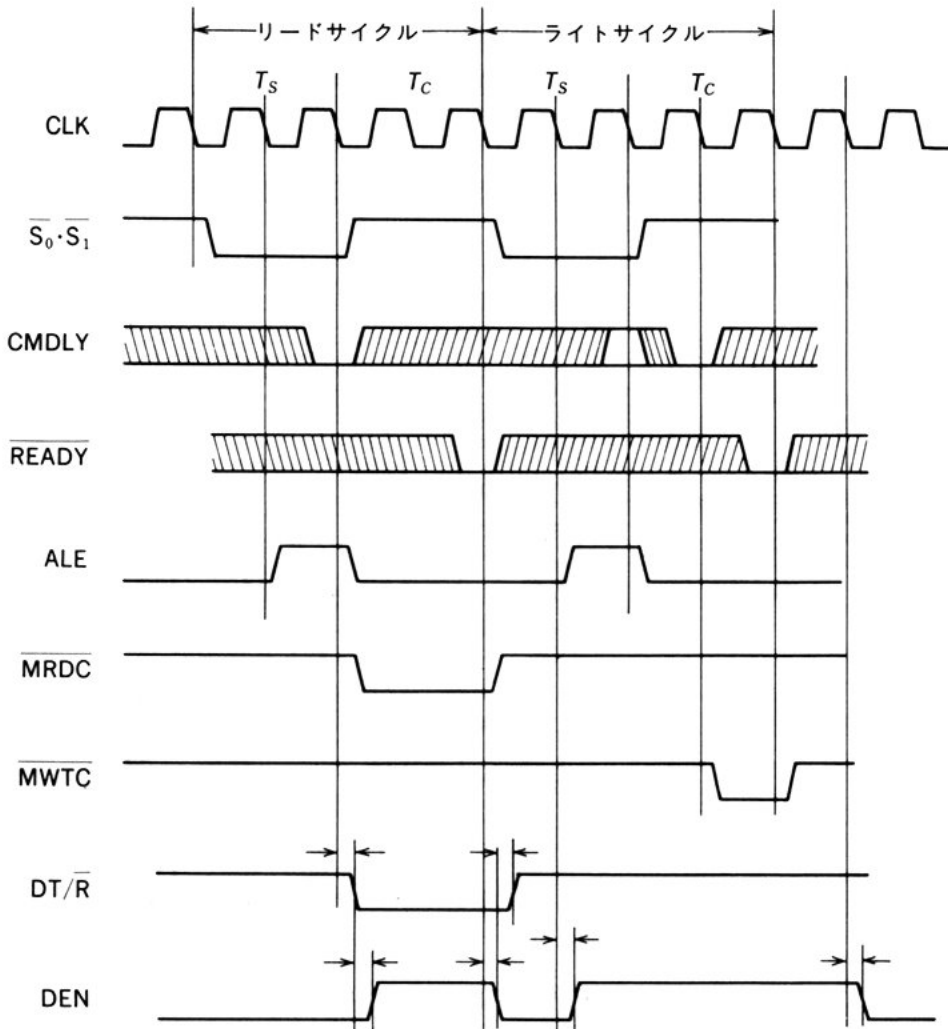



図 8·9　82288 コマンド信号出力タイミング

信号によって制御できる.

メモリリードサイクルとメモリライトサイクルが連続して実行されるときのタイミングチャートを図 8・9 に示す．最初のリードサイクルにおける $\overline{\mathrm{MRDC}}$ は，T_s の立ち下がりにおいて CMDLY が Low であるため T_c において出力されている．次のライトサイクルにおける $\overline{\mathrm{MWTC}}$ は，CMDLY が T_s の立ち下がりにおいて High で，次の T_c の $\phi 1$ の立ち下がりにおいて Low となっているため，1 CLK だけ遅れて出力される．このような CMDLY の制御は $\overline{\mathrm{MRDC}}$，$\overline{\mathrm{MWTC}}$ と同様に $\overline{\mathrm{IORC}}$，$\overline{\mathrm{IOWC}}$，$\overline{\mathrm{INTA}}$ に対しても有効に働く．CMDLY の制御の必要がなければ，CMDLY 端子を GND（接地）に接続すればよい．

82288 は，コマンド信号の他にアドレスラッチ回路へのストローブ信号 ALE，データバストランシーバへの制御信号 $\mathrm{DT}/\overline{\mathrm{R}}$，DEN を出力する.

〔4〕 **80286 CPU 構成**　80286 を使用した CPU 構成は，80286 を中心として，クロックジェネレータ 82284，バスコントローラ 82288，アドレスラッチ，データバストランシーバによって，図 8・10 のように構成される.

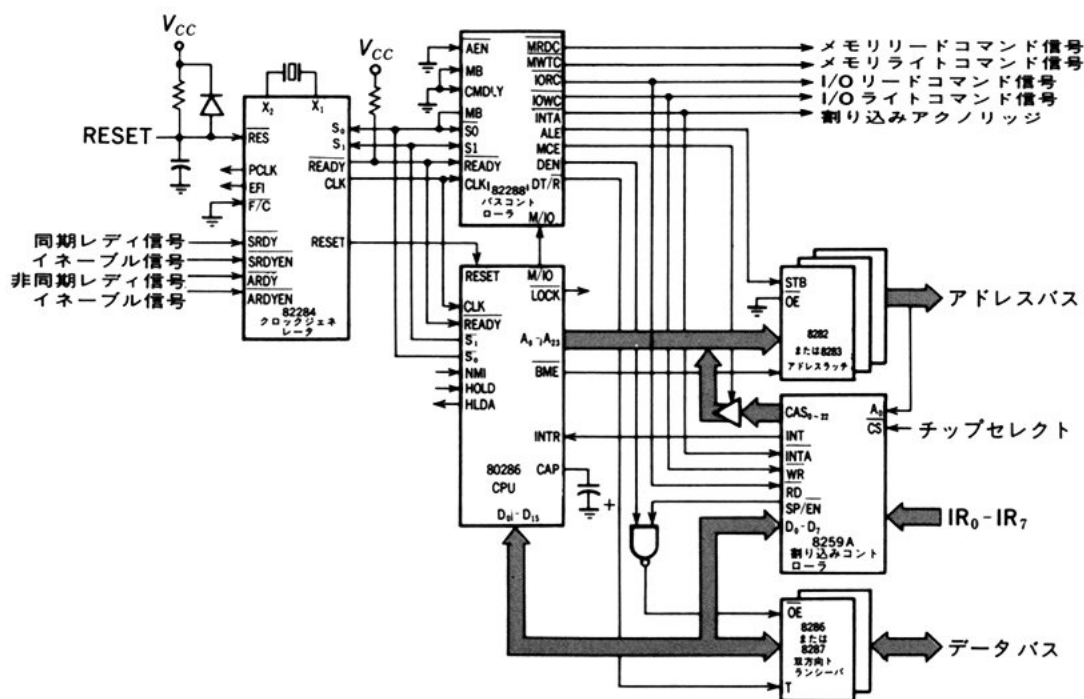



図 8・10　80286 基本構成
(iAPX 286 Hardware Reference Marual (Order No.210760-001), A-34, Fig 31 より引用)

8-2 メモリインタフェース

〔1〕 **基本メモリ構成**　80286 のバス信号とメモリとの接続は，基本的には図 8・11 に示すように 8086 の場合と同じである．すなわち，メモリは偶数アドレスのバンクと奇数アドレスのバンクに分割し，偶数バンクのメモリはデータバス D_0-D_7 に接続する．また，奇数バンクのメモリはデータバス D_8-D_{15} に接続する．アドレスバスには A_1 から A_{23} の信号を使用し，A_0 は偶数バンクのチップイネーブル信号として使用する．また，奇数バンクのチップイネーブル信号には $\overline{BHE}$ を使用する．このように接続することによって，$A_0=0$，$\overline{BHE}=1$ のとき偶数バンクのメモリだけが有効に動作する．また，$A_0=1$，$\overline{BHE}=0$ のときは奇数バンクのメモリだけが有効に動作する．さらに $A_0=0$，$\overline{BHE}=0$ のとき，偶数バンク，奇数バンク両方のメモリが有効に動作する．

図 8・11 をコマンド信号も含めて書き換えると，図 8・12 のようになる．図において，アドレスラッチの STB（ストローブ）信号にバスコントローラ 82288 の ALE を供給しているが，図 8・13 に示すように，ALE 信号は T_s の $\phi 2$ に出力されるので，アドレスバスにアドレス信号が供給されるのは，T_s の $\phi 2$ から，次のバスサイクルの T_s の $\phi 1$ の終わりまでである．メモリからデータをリードする場合について考えると，メモリは T_c の終わりから図 8・13 に示すセットアップ

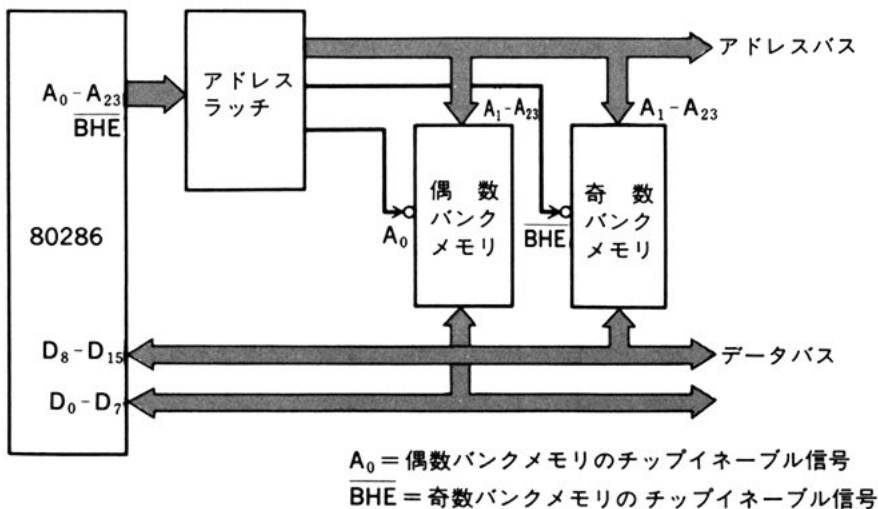



図 8・11　メモリとアドレスバス，データバスの接続

タイム以前に，データをデータバスに出力すればよい．したがって，図に示す T_{ad} はメモリ応答の最大余裕時間であり，t_{ad} より速い応答速度をもつメモリを使用すれば，問題なくデータをリードすることができる．

〔2〕 **特別なストローブ信号を使用したメモリ構成**　しかし，80286 自体はアドレス信号を T_s より以前に出力しているのであるから，図 8・12 に示した回路では，T_s の $\phi 2$ 以前に出力されているアドレス信号を無駄にしている．アドレスラッチの STB 信号をもう少し早いタイミングで供給できれば，図 8・13 に示した

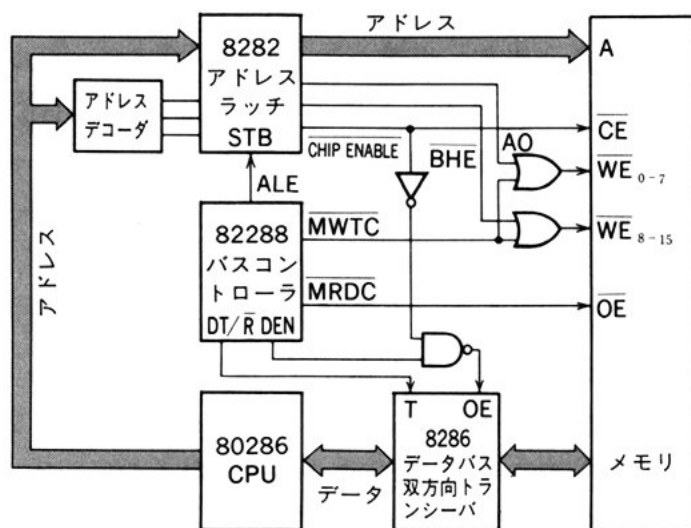



図 8・12　基本的メモリ構成
(iAPX 286 Hardware Reference Manual (Order No.210760-001), 4-7, Fig 4-1 より引用)

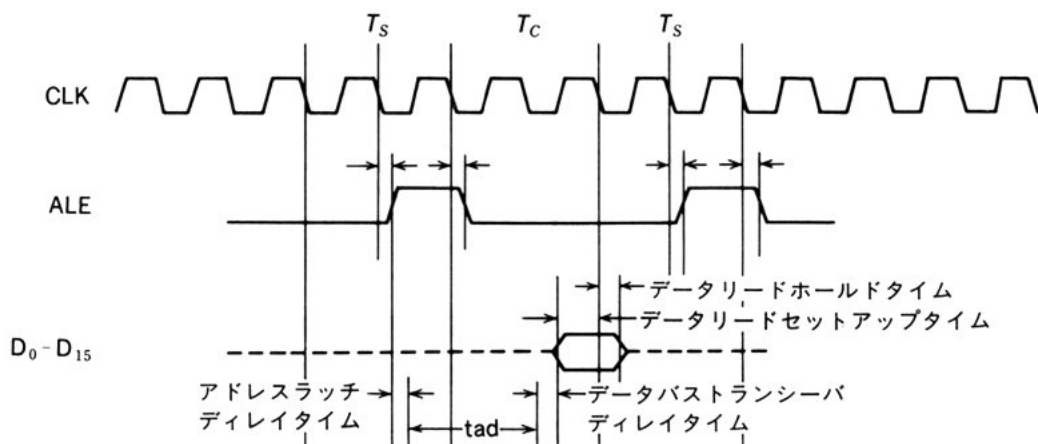



図 8・13　ALE をアドレスラッチのストローブ信号として使用した場合のメモリ応答最大余裕時間

t_{ad} より遅い応答スピードをもつメモリを使用することができる．図 8·14 に示すメモリ構成では，82288 が出力する ALE 信号を使用せずに，$\overline{\text{MRDC}}\cdot\overline{\text{MWTC}}=0$ の条件をアドレスラッチのストローブ信号として使用する．

メモリに対するバスサイクルを実行するとき，82288 は遅くとも T_s の最初に $\overline{\text{MRDC}}$ と $\overline{\text{MWTC}}$ の両方を High にし，T_c の最初に $\overline{\text{MRDC}}$，$\overline{\text{MWTC}}$ のどちら

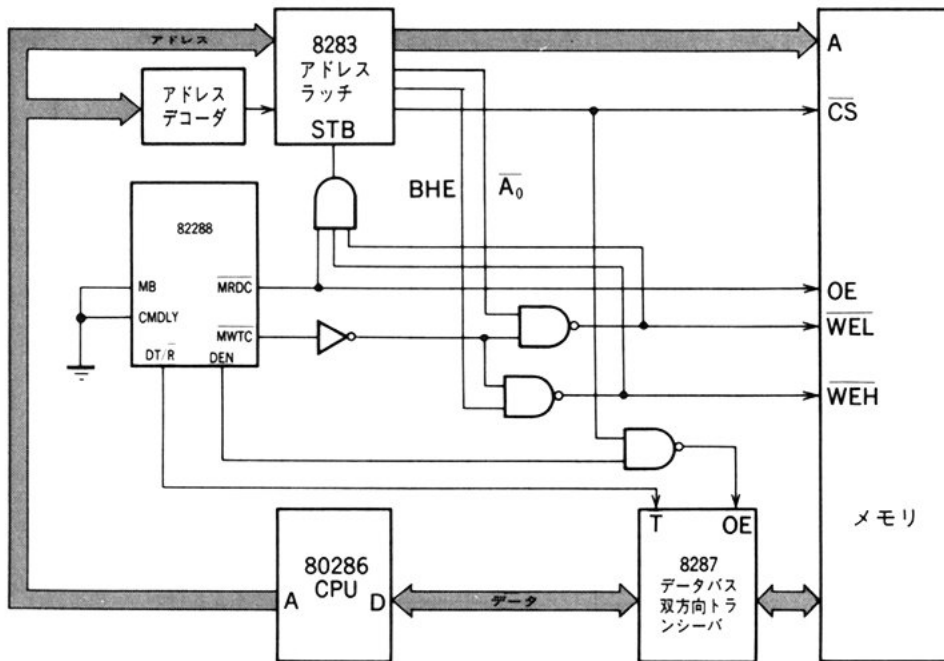



図 8·14 特別ストローブ信号を使用したメモリ構成
(iAPX 286 Hardware Reference Manual (Order No. 210760-001), 4-8, Fig 4-2 より引用)

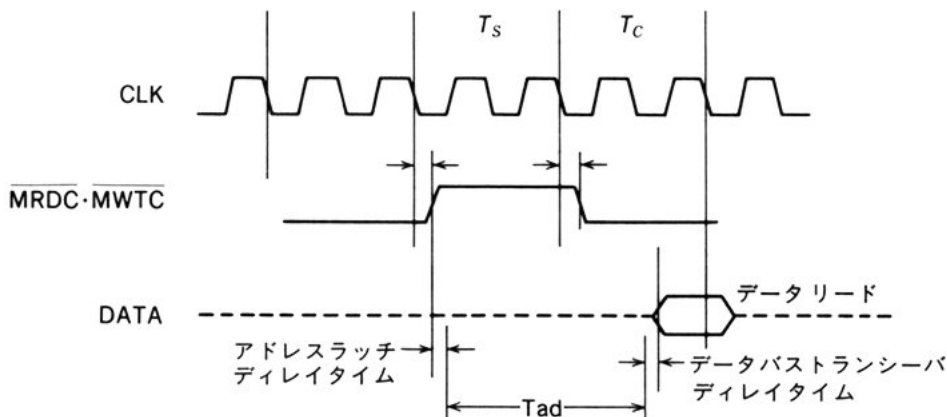



図 8·15 $\overline{\text{MRDC}}\cdot\overline{\text{MWTC}}$ をアドレスラッチのストローブ信号として使用した場合のメモリ応答最大余裕時間

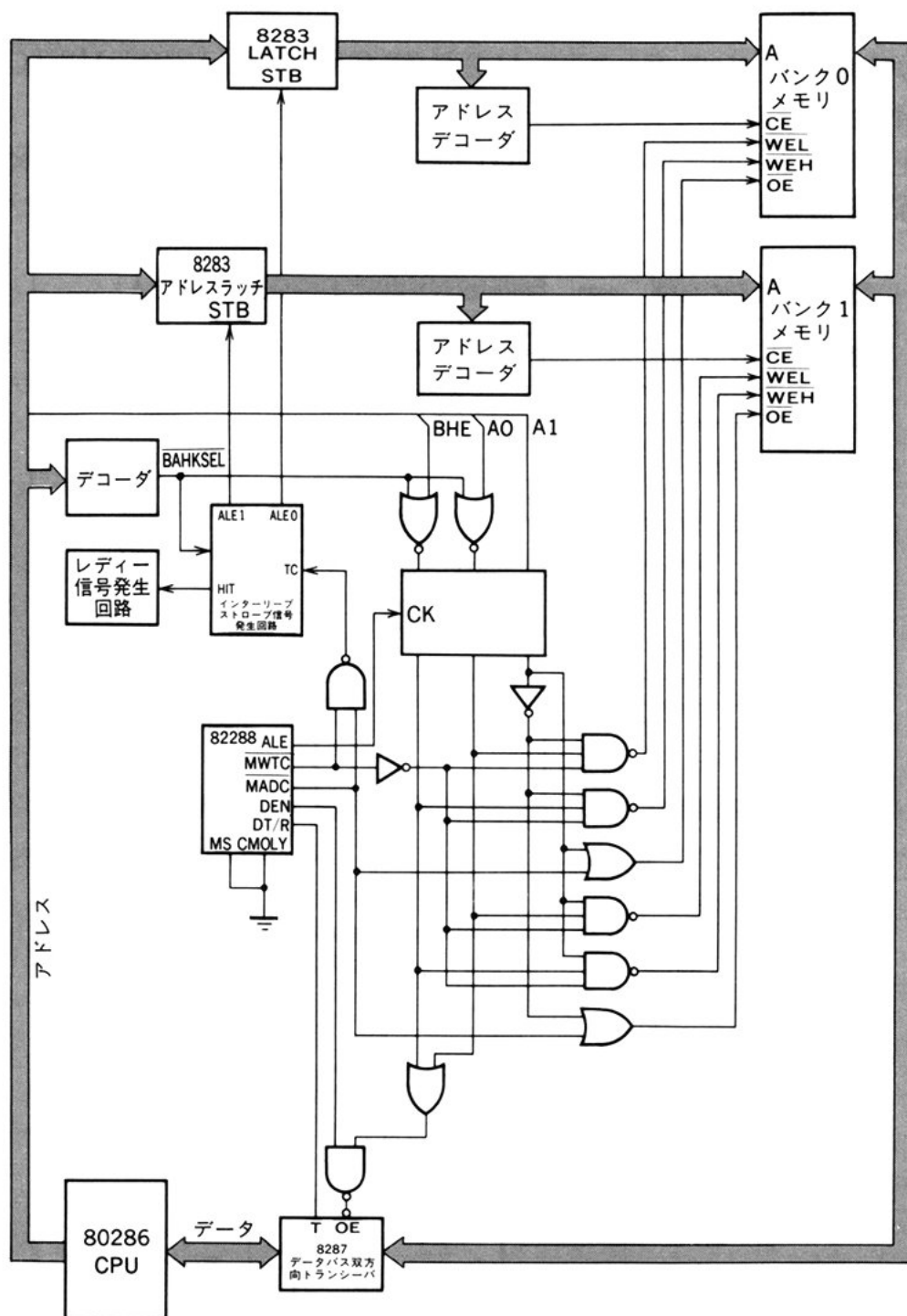



図 8・16 パイプラインアクセスを利用したメモリ構成
(iAPX 286 Hardware Reference Manual
(Order No. 210760-001), 4-10, Fig 4-3 より引用)

か一方を Low にするから $\overline{\mathrm{MRDC}}$ と $\overline{\mathrm{MWTC}}$ の論理 AND をとることによって図 8・15 に示すような信号が作られる.

この構成では，コマンド信号の立ち上がりでアドレスラッチにストローブ信号が供給され，アドレスバスに新しいアドレスが出力される．したがって，図 8・12 に示した基本的メモリ構成より 1 CLK だけ大きなメモリ応答の最大余裕時間 T_{ad} が得られる.

〔3〕 **パイプラインアクセスの利用**　80286 がもつパイプライン的なアドレス出力を有効に利用するために，図 8・16 に示すようなメモリ構成にすることができる．この構成では，メモリをアドレスバスの A_1 の値によって，2 つのメモリバンクに分割し，それぞれのメモリバンクには独立したアドレスバスからアドレスを供給する．A_1 の値が 0, 1, 0, 1, … となるように，80286 がバスサイクルを実行するならば，バンク 0 のメモリがデータの入出力を実行しているときに，次のバスサイクルのアドレスをバンク 1 に送ることができる．また，バンク 1 のメモリの実行終了を待たずに，さらに次のアドレスをバンク 0 に供給することができる．もちろん，80286 は同じバンクに連続してバスサイクルを実行する場合もある．このときはパイプライン的なメモリ参照はできず，バスサイクルにウェイトステートを挿入する必要がある．しかし，プログラムの平均的な実行状況において，同じメモリバンクへの連続的な参照の回数は，全体のメモリ参照の 7％程度である．したがって，図 8・16 に示したようなメモリ構成の効果は十分にある.

8-3 I/O インタフェース

I/O インタフェースは CPU と周辺装置の間で，データ変換を行う回路である．コンピュータと周辺装置との間のデータ伝送には，セントロニクス，RS-232-C などのいくつかの標準的な方法がよく使用される．また，これらの方法でデータの入出力を行う I/O インタフェースも，半導体各社で製造されている．

図 8·17 に I/O インタフェースの特徴を表す簡単なモデルと，80286 のバスとの接続を表す．I/O インタフェースは内部に，データを一時的に保存するためのレジスタまたは I/O インタフェース自身の動作モードを決めるためのレジスタなどをもち，これらのレジスタはアドレスデコーダの設計によって，特定の I/O アドレスが与えられている．80286 では，IN 命令，OUT 命令のオペランドに I/O アドレスを指定することによって，I/O インタフェース内部の各レジスタからデータを入力したり，または逆にデータを書いたりすることができる．このとき実行されるバスサイクルは，メモリのリード，ライトで実行されるものと，基本的には同じである．80286 は A_0-A_{15} に I/O アドレスを出力し，また $\overline{S_0}$, $\overline{S_1}$, M/$\overline{IO}$

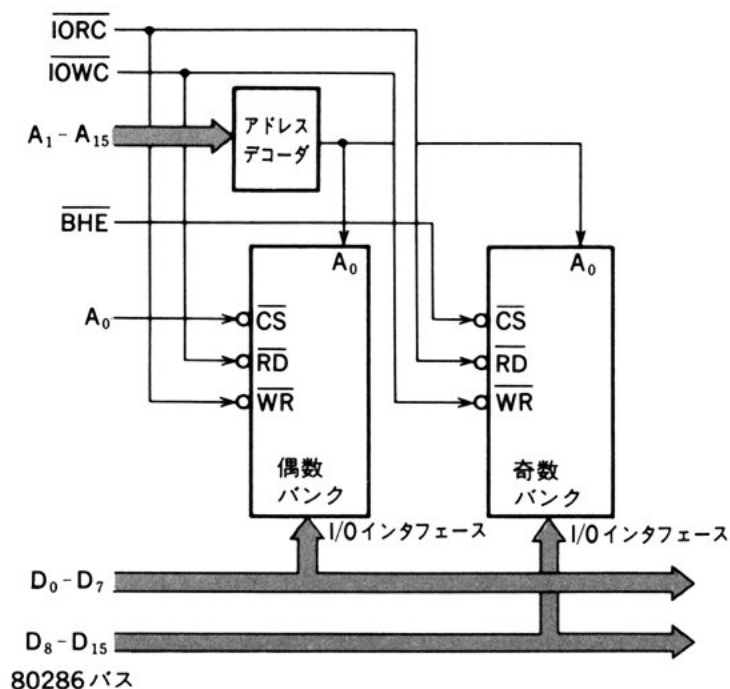



図 8·17　I/O インタフェース

にI/Oのリードまたはライトであることを示すステータス信号を出力する．I/Oアクセスのとき，A_{16}-A_{23}は0になっている．

たとえば，図8·17に示す回路例ではI/Oインタフェース内部に2種類の8ビットレジスタがあり，I/OインタフェースのA_0端子を0にするか，1にするかによって，どちらかのレジスタが選択されるものとする．

I/Oインタフェースとバスの接続も，メモリを接続する場合と基本的には変わらない．ただし，IN命令，OUT命令によって，リードまたはライトが実行できるように，コマンド信号として$\overline{IORC}$，$\overline{IOWC}$に応答するように接続する．その他はメモリの接続と同様に，データバスの下位バイトに接続するI/Oインタフェースの$\overline{CS}$（チップセレクト）端子には，アドレスバスのA_0を接続し，データバスの上位バイトに接続するI/Oインタフェースの$\overline{CS}$端子には，$\overline{BHE}$を接続する．したがって，データバスの下位バイトに接続するI/Oインタフェースの内部レジスタはすべて偶数アドレスをもち，逆にデータバスの上位バイトに接続したI/Oインタフェースの内部レジスタはすべて奇数アドレスをもつ．

このように，80286のバスとI/Oインタフェースの接続は，コマンド信号の種類が異なる他は，メモリの接続の場合と変わるところはない．したがって，I/Oインタフェースにも，$\overline{MRDC}$，$\overline{MWTC}$のコマンド信号と，A_0-A_{23}までのアドレスをデコードした信号を供給してもかまわない．このとき，I/Oインタフェース内部のレジスタのリード，ライトは，IN命令またはOUT命令ではなく，MOV命令などを直接に使用することができる．I/Oインタフェースをメモリと見たてて，バスに接続することを**メモリマップトI/O**（メモリに配置されたI/O）と呼ぶ．

8-4 ローカルバス制御

図 8·18 に**ローカルバス**の構成を示す．ローカルバスは一般に 1 つの CPU によって制御され，他の CPU との間で共有されないようなバスである．しかし，ローカルバスにも **DMA コントローラ**のようなバスを能動的に制御する CPU 以外のバスマスタをもつ場合がある．DMA コントローラは，メモリ-メモリ間，またはメモリ-I/O 間のデータ転送を行うコントローラであり，大量のデータを連続的に転送するとき有効である．

DMA コントローラは CPU から送られるコマンドまたは他の回路から送られるリクエスト信号によって動作を開始する．このとき，CPU から一時的にバスを預かるような制御をしなければ，DMA コントローラと CPU との間で信号の衝突が発生する．80286 の場合，**HOLD 信号**と **HLDA 信号**によって，ローカルバスの制御を実行する．たとえば，DMA コントローラが HOLD 信号を High にしたとき，80286 は D_0-D_{15}，A_0-A_{23}，$\overline{BHE}$，$\overline{S_0}$，$\overline{S_1}$，M/$\overline{IO}$，COD/$\overline{INTA}$，$\overline{LOCK}$，PEREQ などの，バス制御に関係する端子をハイインピーダンス状態にしてから，HLDA 端子に High の信号を出力する．この後，DMA コントローラが 80286 に代わってバスを制御する．

もちろん，DMA コントローラが一連の処理を実行し終ったとき，HOLD 信号を Low に戻す．このとき，80286 は HLDA 信号を Low に戻してから，再びバス制御を取り返すことができる．

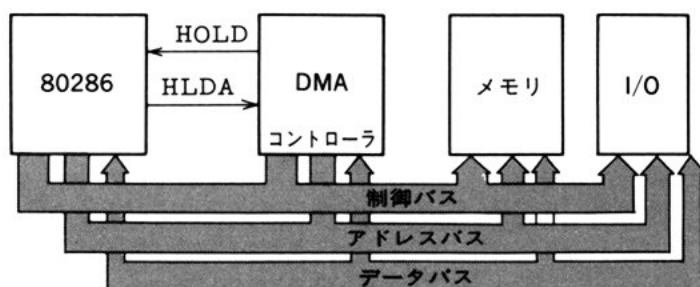



図 8・18　ローカルバス

8-5 システムバス制御

コンピュータの負荷が大きくなった場合，複数のCPUに負荷を分散させることによって，問題を解決することができる．たとえば，本来の計算処理を実行するCPUと，入出力処理を専門に実行するCPUに分割する．このようなとき，**システムバス**または**標準バス**と呼ばれる，CPUの特性には関係ない標準のバスを用意することが有効である．

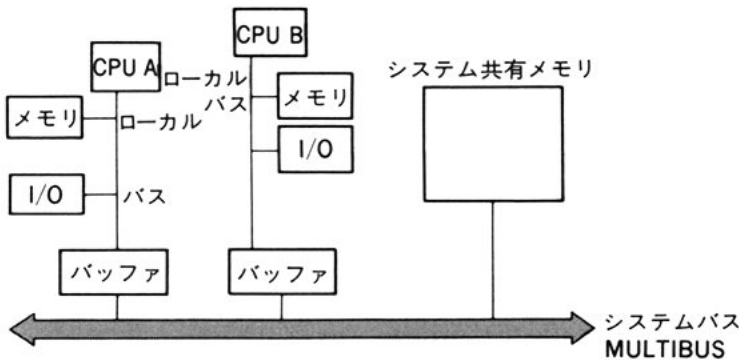
システムバスの構成は図8·19に示すようになる．システムバスには，複数のCPUを接続することができ，それぞれのCPUはローカルバスを使用して，独立に動作することができる．また，システムバスに接続されるメモリは，システ

注） MULTIBUSはインテル標準のシステムバスである．

図 8・19　システムバス

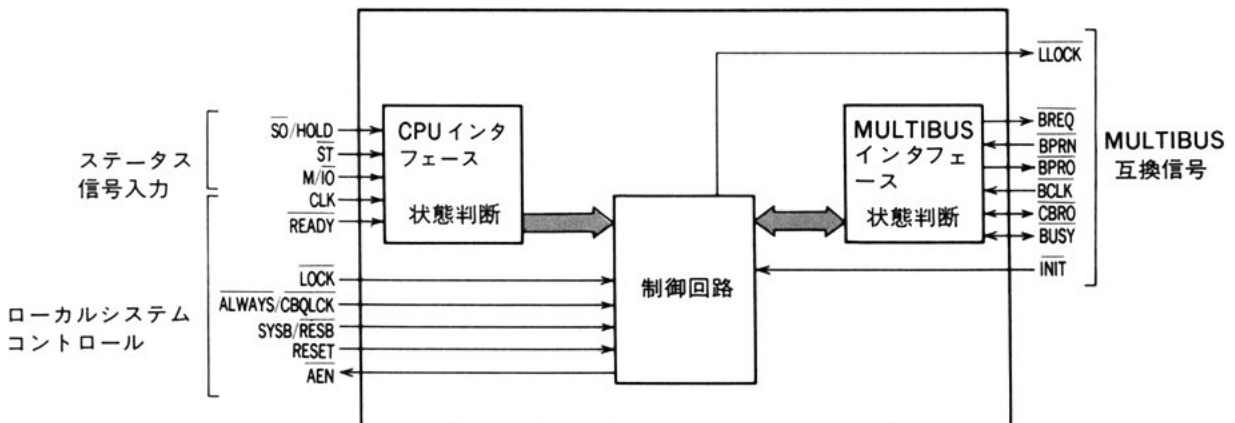



図 8・20　82289 バスアービタ

(Microsystem Components Handbook Microprocessors Vol. 1 (Order No. 230843-003), 4-167, Fig 1より引用)

ムバス上の任意の CPU から共有される共有メモリとして働く．たとえば，CPU A によって処理したデータを CPU B において，さらに別の処理を実行するためデータを渡したいようなとき，CPU A が共有メモリにデータを書いた後で，CPU B が共有メモリからデータを読むという方法で実現できる．

このように，システムバスは複数の CPU から共有されるものであるから，システムバス上で信号の衝突が発生しないような柔軟な制御が必要である．システムバスの制御を行う回路を**バスアービタ**と呼ぶ．80286 のバスアービタには **82289** を使用することができる．82289 のブロック図を図 8·20 に示す．

バスアービタは，システムバスに接続される CPU ボード上に 1 個ずつ配置し，バスアービタ相互の接続によって決まる優先順位に従って，システムバスを要求する CPU の中で，最も優先順位の高いものにだけシステムバスの使用を許す．システムバスを獲得できなかったバスアービタは，システムバスへのバスバッファ出力をハイインピーダンス状態にし，CPU はウェイト状態になるように 82284 を制御する．

9. 数値演算コプロセッサ80287

浮動小数点演算を高速で実行することは，マイクロコンピュータにとっても重要な働きである．しかし，そのような機能を必要としない場合もあるから，浮動小数点演算機能を CPU に内蔵することは不経済である．そこで，8086 のときに**コプロセッサ**という概念が生まれた．8086 だけを使用すれば，ワードタイプまでの整数演算しかできないが，コプロセッサ 8087 を追加することによって，浮動小数点演算も実行できるようになる．コプロセッサは単独では動作することができない．80286 の場合は，数値演算コプロセッサ 80287 をもつ．80287 の内部は 8087 と同様である．ただし，80287 は，ホストとなる CPU（80286 または 80386）との接続方式が 8087 と 8086 の接続の場合と異なる．したがって，8087 の応用プログラムは，80287 においてもほとんど変更することなく動作する．

9-1　80287 のアーキテクチャ

図 9·1 に 80287 のレジスタ構成を示す．**レジスタスタック**と呼ばれる 80 ビットのレジスタが 8 本ある．レジスタスタックは，演算の対象となる浮動小数点データを格納するレジスタであるが，各レジスタを識別するためのレジスタ名はもたず，メモリのアドレスのように 0 から 7 までのアドレスが与えられている．さらに，これらのレジスタスタックは，ちょうどメモリのスタックのように使用される．図に示すステータスレジスタ中の 3 ビットの TOP フィールドの値によって，現在のスタックトップのレジスタを表す．スタックトップのレジスタがアキュムレータとなる．このように，80287 はいわゆるスタックマシンとして使用するこ

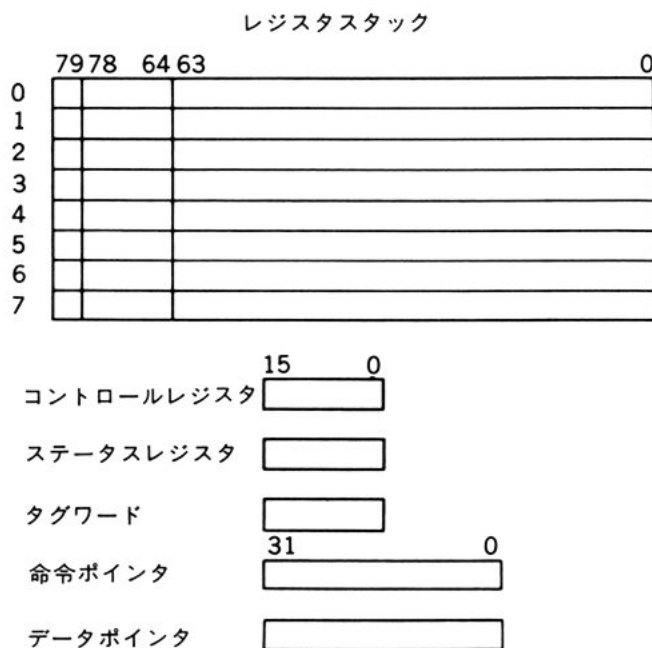



図 9·1　80287 のレジスタ構成

図 9·2　タグワード

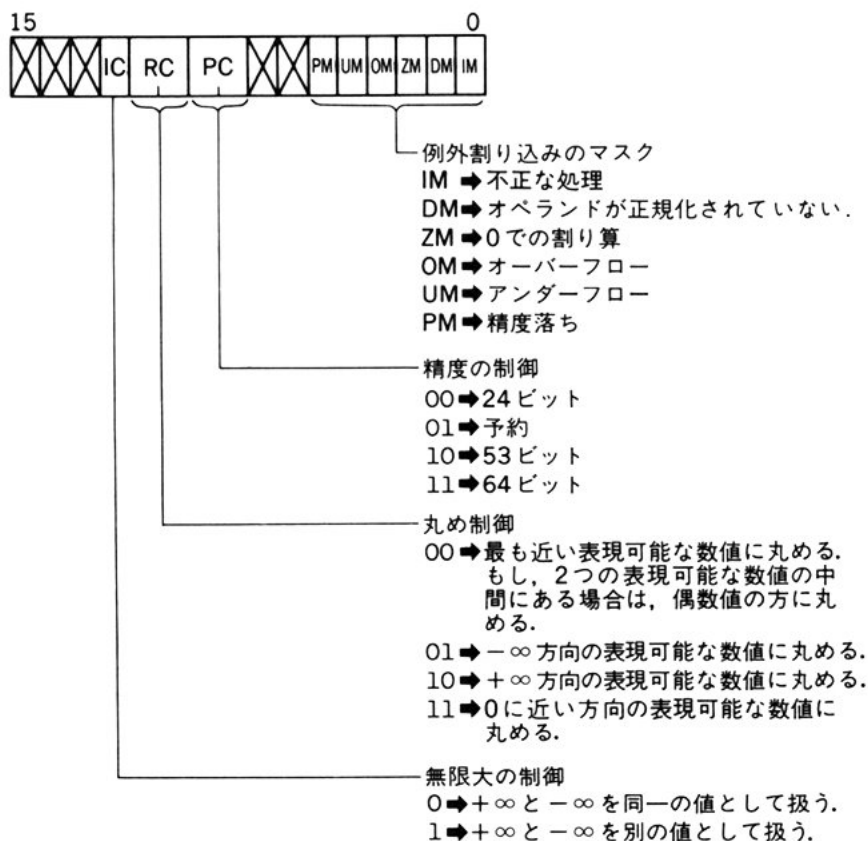



図 9・3 コントロールレジスタ

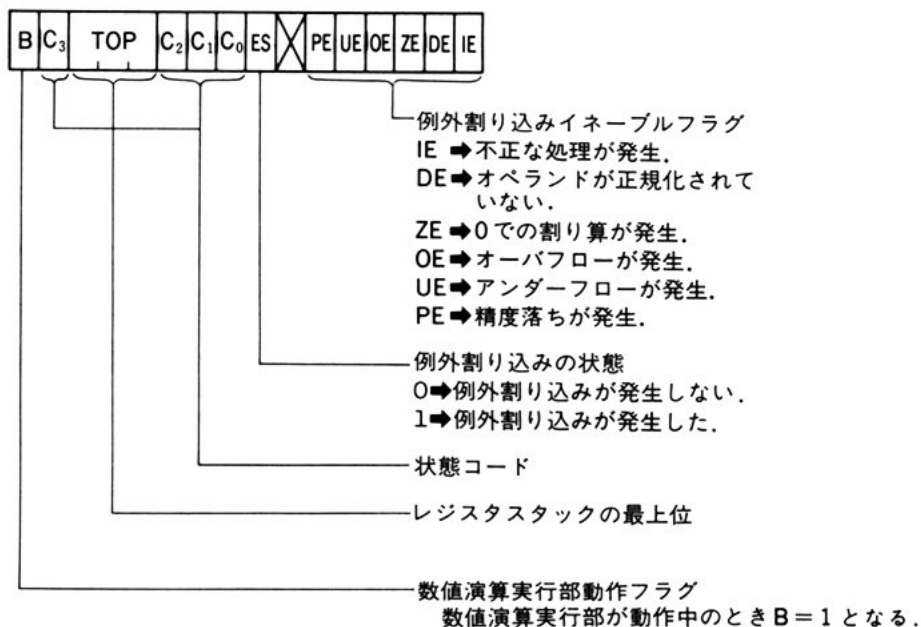
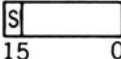
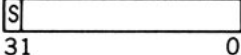
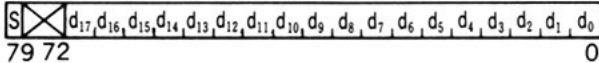
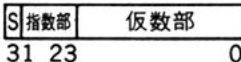



図 9・4 ステータスレジスタ

とができる．

タグワードは，図9·2に示すように16ビットのレジスタで，2ビットの各フィールドがそれぞれレジスタスタックに対応し，レジスタスタックの状態を表す．タグワードは，ほとんど80287の内部動作において，レジスタスタックの値の評価を高速化するために使用されるが，ユーザプログラムから直接にタグの値を読んだり，変更することも可能である．

コントロールレジスタは，80287の動作を制御するレジスタである．コントロールレジスタの構成を図9·3に示す．ビット0からビット5までは数値演算の実行における例外割り込みのマスクである．これらのマスクビットを1にしておけば，図9·3に示す状況が発生しても，80287は80286への割り込み要求信号を発生しない．PCの2ビットはレジスタスタックの演算結果の精度を制御する．PCの値によって，仮数部の有効桁を24ビット，53ビット，64ビットのどれかに決めることができる．また，ICの値によって無限大数の扱いを制御することもできる．すなわち，IC=0のとき $+\infty$ と $-\infty$ は同じ値として扱うのに対して，IC=1のとき $+\infty$ と $-\infty$ を別の値として扱う．なお，80287は80ビットの特別なコードで，$+\infty$ および $-\infty$ を表現することができる．

ステータスレジスタは80287の処理状況を表す，80286のFLAGに相当するレジスタである．図9·4にステータスレジスタの構成を示す．ビット0からビット5は，数値演算処理において例外割り込みが発生したときに1になる．各ビットの例外割り込みの種類は，コントロールレジスタのビット0からビット5までのマスクビットに対応している．80287の例外処理に対しては，コントロールレジスタでマスクすることによって，割り込み要求信号を発生させずに，ステータスレジスタを読み取って，80287の状態を知ることもできる．ESはマスクされていない例外割り込みが発生したときに1になる．C_0, C_1, C_2, C_3 はデータの比較命令の結果などが残る．TOPの3ビットはレジスタスタックの最上位のレジスタを指定する．Bビットは80287が実行中のとき1になる．

9-2 データの表現

80287は図9・5に示すようなデータを表すことができる．すなわち，メモリに定義された図に示す各データをレジスタスタックに転送することができる．このとき，すべてのデータは，**テンポラリリアル**と呼ばれる80ビットの大きさの浮動小数点表現に自動的に変換されてから，スタックレジスタに代入される．したがって，スタックレジスタ上では，データはすべてテンポラリリアルの形で表現される．逆に，スタックレジスタの値をメモリに記録するとき，図に示す任意の形式に変換することができる．

整数は最上位ビットが符号ビットとなり，マイナスの値は2の補数で表される．大きさは16ビット，32ビット，64ビットの整数がある．また，パックトデシマル

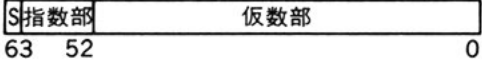
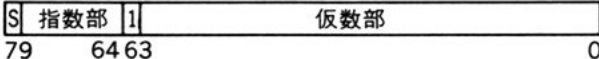



図 9・5　80287のデータ

は特別な形式で表す整数で，10 進数の 1 桁を 4 ビットで表す．80 ビットの下位 72 ビットで 17 桁までの 10 進整数を表すことができる．最上位ビットは符号ビットとなる．

浮動小数点データには，大きさが 32 ビット，64 ビット，80 ビットの 3 種類のものがある．スタックレジスタ中のデータは，すべて 80 ビットのテンポラリリアルで表現される．メモリからスタックレジスタにデータをリードするとき，またはスタックレジスタの値をメモリにライトするとき，80287 はテンポラリリアルと他のデータタイプの間のデータ変換を自動的に実行する．

IEEE 754 と 80287

コンピュータの互換性の問題の中に，浮動小数点データのフォーマットの問題がある．同じ FORTRAN のプログラムを実行しても，コンピュータの機種によって浮動小数点データの表現が異なり，丸め誤差の影響によって出力データまたはその動作に違いが発生する場合がある．そこで，浮動小数点データのフォーマットを標準化するために IEEE 754 浮動小数点データ標準化案がある．8087 と同様に，80287 も IEEE 754 に準じた単精度および倍精度浮動小数点データを扱う．

しかし 8087, 80287 内部では，データはすべて 80 ビットの大きさのテンポラリリアルと呼ばれる 8087，80287 の内部表現に変換されて扱われる．32 ビット長の単精度または 64 ビット長の倍精度浮動小数点データをテンポラリリアルに拡張することは，演算途中のオーバフローに対するマージンとなる．たとえば，簡単のために 10 進 2 桁までのデータを扱うことができるコンピュータがあるとして，10×10÷2 の演算を実行すると結果は 50 であるが，途中で 100 というデータが発生しこのコンピュータの扱えるデータの大きさを超えてしまう．演算順序を変更するなどの方法でオーバフローを避けることもできるが，最も簡単な方法はこのコンピュータを 10 進 3 桁までのデータを扱えるようにすることである．

また，8087，80287 は整数と BCD を扱うこともできるが，これらのデータも 8087，80287 内部ではテンポラリリアルに変換して扱われることに注意する．

9-3 レジスタスタックの基本的使用

〔1〕 **レジスタスタックの構成**　レジスタスタックは，図9･6(a)に示すように80ビットの8本のレジスタで，0から7までの番地が与えられる．ステータスレジスタTOPの3ビットの値が，スタックレジスタの現在の最上位のレジスタを表す．最上位のスタックレジスタは，アキュムレータとして80287の演算処理の中心となるデータを保存する．最上位のスタックレジスタを80287の命令のオペランドに指定するときには，STと書く．80287は，純粋のスタックマシンではなく，スタックレジスタの最上位以外のレジスタを任意に参照することもできる．このとき，ST(1)からST(7)の名前で，それぞれのスタックレジスタを指定することができる．もし，STが0番のスタックレジスタであれば，1番のスタックレジスタはST(1)，また7番のスタックレジスタはST(7)となる．また，図9･6(b)に示すように，7番のスタックレジスタがSTであれば，0番のスタックレジスタはST(1)となり，6番のスタックレジスタがST(7)となる．

〔2〕 **レジスタスタックへの代入**　レジスタスタックに値を代入するために

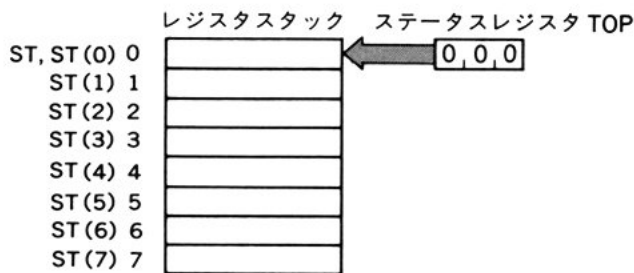
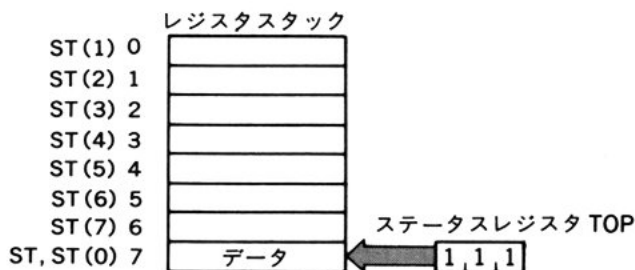



図 9・6　レジスタスタックの使用

表 9・1 FLD命令の種類

オペコードニーモニック	オペランド	動作
FLD	DWORD タイプメモリ	メモリに定義したショートリアルデータをテンポラリリアルに変換してスタックレジスタにPUSHする.
	QWORD タイプメモリ	メモリに定義したロングリアルデータをテンポラリリアルに変換してスタックレジスタにPUSHする.
	TBYTE タイプメモリ	メモリに定義したテンポラリリアルデータをスタックレジスタにPUSHする.
	ST (i) (ここでi=0~7)	スタックレジスタ ST (i) の値をスタックレジスタの最上位にPUSHする.
FILD	WORD タイプメモリ	メモリに定義したワード整数データをテンポラリリアルに変換して, スタックレジスタにPUSHする.
	DWORD タイプメモリ	メモリに定義したショート整数データをテンポラリリアルに変換してスタックレジスタにPUSHする.
	QWORD タイプメモリ	メモリに定義したロング整数データをテンポラリリアルに変換してスタックレジスタにPUSHする.
FBLD	TBYTE タイプメモリ	メモリに定義したパックトデシマルデータをテンポラリリアルに変換してスタックレジスタにPUSHする.

表 9・2 FST命令

オペコードニーモニック	オペランド	動作
FST ESTP	DWORD タイプメモリ	レジスタスタックの最上位にあるデータをショートリアルに変換して, メモリに保存する.
	QWORD タイプメモリ	レジスタスタックの最上位にあるデータをロングリアルに変換して, メモリに保存する.
	TBYTE タイプメモリ	レジスタスタックの最上位にあるデータをテンポラリリアルのまま, メモリに保存する.
	ST (i) (ここでi=0~7)	レジスタスタックの最上位にあるデータを他のレジスタスタックに保存する.
FIST FISTP	WORD タイプメモリ	レジスタスタックの最上位にあるデータをワード整数に変換して, メモリに保存する.
	DWORD タイプメモリ	レジスタスタックの最上位にあるデータをショート整数に変換して, メモリに保存する.
	QWORD タイプメモリ	レジスタスタックの最上位にあるデータをロング整数に変換して, メモリに保存する.
FBSTP	TBYTE タイプメモリ	レジスタスタックの最上位にあるデータをパックトデシマルに変換して, メモリに保存する.

オペコードニーモニックで最後にPの付いているものは, 動作の最後にレジスタスタックの最上位のTAGを11Bにして, TOPの値を1だけ増分する操作を含む.

は，FLD 命令を使用する．FLD 命令は

FLD　オペランド

のように書き，ステータスレジスタの TOP から 1 だけ減算してから，TOP が指定するレジスタスタックに，オペランドで指示したデータを代入する．このとき，オペランドのデータは自動的にテンポラリリアルに変換される．このように，FLD 命令の動作はメモリへの PUSH 命令とよく似ている．また，FLD 命令には，オペランドで指定するデータのタイプによって，表 9･1 に示す種類がある．

80287 のアセンブリ言語の命令では，ロングリアルとロングインテジャのデータは，どちらも 64 ビットの大きさであるから，オペランドのタイプだけでは区別できない．また，パックトデシマルとテンポラリリアルの場合も，どちらも 80 ビットの大きさであるから，オペランドのタイプだけでは区別できない．そこで，80287 の命令のニーモニックは表に示したように，F の次に I または B を書くことによって，リアルタイプ，インテジャタイプ，パックトデシマルのデータを区別する．図 9･6 (a) に示したように，TOP の値が 0 のとき，FLD 命令を使用して，メモリまたは他のレジスタスタックから，レジスタスタックの最上位にデータを代入すれば図 (b) に示すようになる．すなわち，TOP の値は，3 ビットの演算でキャリーを無視するから 0 から 1 を減算することによって 7 になり，7 番のレジスタスタックが新しい ST になる．

このように，FLD 命令によって，レジスタスタックへのデータの PUSH を繰り返し実行したとき，レジスタスタックに代入できるデータの数は最大 8 までである．レジスタスタックにデータを代入するためには，対応する TAG の値がそのレジスタスタックがデータをもっていないことを示す 11B の値でなければならない．もし，すべてのレジスタスタックがデータをもつときに，FLD 命令を実行してもデータを代入することはできない．

〔3〕 **レジスタスタックのストア**　FLD 命令とは逆に，レジスタスタックの最上位のデータを，メモリまたは他のレジスタスタックに書くためには FST 命令を使用する．FST 命令は

FST　オペランド

のように書く．FST 命令の場合も，扱うデータタイプによって表 9･2 に示す種類がある．FST 命令を実行したとき，レジスタスタックの最上位のデータが，命令のニーモニックとオペランドタイプの組み合わせによって決まるデータに変換

されてから，メモリまたは他のレジスタスタックに書かれる．

FST 命令の場合は，オペコードのニーモニックの最後に P が付いているものと，付いていないものがある．最後に P が付いている FST 命令は POP 操作を含む．すなわち，命令の最後でレジスタスタックの最上位の TAG を 11B に戻してから，TOP の値に 1 を加算する．したがって，図 9·6 (b) の状態で FSTP 命令を実行したとき，レジスタスタックの状態は図 (a) に示すようになる．FST 命令のニーモニックの最後に P が付いていない命令は上のような POP 操作を含まないから，レジスタスタックの最上位は変化しない．

スタックマシン

スタックマシンとは CPU 内部のレジスタがメモリのスタッフのような構成になっているコンピュータのことをいう．演算処理はレジスタスタックの最上位にある 1 つまたは 2 つのレジスタに対して実行され，一般に命令のオペランドを指定する必要はない．演算結果は最上位のレジスタに残る．

このように，スタックマシンでは，最も新しいデータがレジスタスタックの最上位に位置し，あまり使用されないデータはスタックの底に残っていくことになる．

80287 は基本的にスタックマシンの構成をもっているが，完全に純粋なスタックマシンではない．ST (3) などのレジスタ名を用いて特定のレジスタをオペランドに明示する命令も備えている．ただし，ST (i) というレジスタ名はスタックトップの位置によって指定されるレジスタが相対的に決まることに注意する必要がある．

9-4 演算命令と関数命令

〔1〕 **基本演算命令**　80287は，加減乗除の基本的な算術演算命令と各種の関数命令をもつ．表9·3に基本的な算術演算命令を示す．算術演算命令では，オペランドをもたないもの，メモリオペランドを1つだけ指定するもの，そして2つのレジスタスタックをオペランドに指定するものがある．しかし，すべての演

表 9・3　算術演算命令

オペコードニーモニック			オペランド	動　作
加算	FADD		オペランドなし メモリ ST, ST(i) ST(i), ST	FADDP ST(1),ST と同じ ST ← ST + メモリ ST ← ST + ST(i) ST(i) ← ST(i) + ST
	FIADD		メモリ	ST ← ST + メモリ
	FADDP		ST(i), ST	ST(i) ← ST(i) + ST, レジスタスタックの POP
減算	FSUB	FSUBR	オペランドなし メモリ ST, ST(i) ST(i), ST	FSUB は FSUBP ST(1),ST と同じ FSUBR は FSUBRP ST(1),ST と同じ ST ← ST − メモリ ST ← ST − ST(i) ST(i) ← ST(i) − ST
	FISUB	FISUBR	メモリ	ST ← ST − メモリ
	FSUBP	FSUBRP	ST(i), ST	ST(i) ← ST(i) − ST, レジスタスタックの POP
乗算	FMUL		オペランドなし メモリ ST, ST(i) ST(i), ST	FMULP ST(1),ST と同じ ST ← ST × メモリ ST ← ST × ST(i) ST(i) ← ST(i) × ST
	FIMUL		メモリ	ST ← ST × メモリ
	FMULP		ST(i), ST	ST(i) ← ST(i) × ST, レジスタスタックの POP
除算	FDIV	FDIVR	オペランドなし メモリ ST, ST(i) ST(i), ST	FDIV は FDIVP ST(1),ST と同じ FDIVR は FDIVRP ST(1),ST と同じ ST ← ST/メモリ ST ← ST/ST(i) ST(i) ← ST(i)/ST
	FIDIV	FIDIVR	メモリ	ST ← ST/メモリ
	FIDIVP	FDIVRP	ST(i), ST	ST(i) ← ST(i)/ST, レジスタスタックの POP

オペコードニーモニックの最後に R のついた命令は逆演算となる．たとえば，FSUBR ST(1)，ST の動作は，ST(1) ← ST − ST(1) となり，また，FDIVR ST,ST(2) は ST ← ST(2)/ST となる．

算命令において，処理の対象の一方は必ず最上位のレジスタスタック ST である．オペランドにメモリを指定した命令は，メモリに定義されたデータをテンポラリリアルに変換した値と，ST の間で演算が行われる．さらにメモリオペランドをもち，オペコードニーモニックの F の次に I が付いている命令は，メモリに定義されたワード整数，またはショート整数をテンポラリリアルに変換してから，ST との間で演算を実行する．また，メモリオペランドをもち，オペコードのニーモニックの F の次に I が付いていない命令は，メモリに定義されたショートリアルまたはロングリアルをテンポラリリアルに変換したデータと ST との間で演算を実行する．演算命令のメモリオペランドに，テンポラリリアル，パックトデシマル，ロング整数を指定することはできない．

減算と除算のオペコードニーモニックの最後に R が付いている命令は逆演算命令となる．たとえば

```
FSUB ST,ST(2)
```

の命令は，レジスタスタックの最上位 ST から ST (2) を減算した結果を ST に代入する．これに対して

```
FSUBR ST,ST(2)
```

の命令は，ST (2) から ST を減算した結果を ST に代入する．このように逆演算命令は，被減数と減数とが逆になる．除算の場合も同様に，オペコードニーモニックの最後に R が付いた命令は逆演算になり，被除数と除数が逆になる．

演算命令においても，オペコードニーモニックの最後に P を付けた命令があり，これは命令の最後にレジスタスタックの POP を実行する．POP 付きの演算命令のオペランドは，必ず両方のオペランドがレジスタスタックであり，右側のオペランドは ST を指定する．

演算命令の中に，オペランドをもたない命令がある．この命令は典型的なスタックマシンの命令であり，演算は必ず ST と ST (1) のレジスタスタックの最上位にある 2 つのレジスタに対して実行され，演算結果を ST (1) に代入してから，最後にレジスタスタックの POP を 1 回だけ実行する．すなわち，レジスタスタックの最上位の 2 つのレジスタが，それらの演算結果をもつ 1 つの最上位のレジスタスタックに置き換わることになる．

〔2〕 **関数命令**　80287 は，加減乗除の四則演算命令の他に表 9・4，表 9・5 に示す関数命令をもつ．関数命令はすべてオペランドをもたない．処理の対象は

表 9・4 80287 基本関数

オペコードニーモニック	動　作
FSQRT	ST ← $\sqrt{ST}$
FSCALT	ST ← ST・$2^{ST(1)}$
FPREM	ST ← ST − ST(1) 上の演算の結果 もし，ST < ST(1) であれば $C_2=0$ また，ST ≧ ST(1) であれば $C_2=1$
FRNDINT	ST の値を整数にまるめる．このとき，小数点以下の数値の扱いはコントロールレジスタの RC によって制御される．
FXTRACT	ST の値を指数部と仮数部に分割し，指数部を ST に代入し，仮数部の値をレジスタスタックにさらに PUSH する．
FABS	ST ← |ST|
FCHS	ST の符号を反転させる．

表 9・5 80287 超越関数

オペコードニーモニック	動　作
FPTAN	ST に定義した θ に対して，Y/X＝tanθ となるような X, Y を求め，Y を ST に代入してから，X をレジスタスタックに PUSH する．したがって，実行後 X が ST に，Y が ST(1) に残る．ただし，θ は 0≦θ≦π/4 でなければならない．
FPATAN	ST に定義した値 X, ST(1) に定義した値 Y に対して，θ＝arc tan(Y/X) となるような θ を求め，ST(1) に代入してから，レジスタスタックを POP する．したがって，ST に θ が残る．ただし，0≦Y<X<∞ でなければならない．
F2XM1	ST に定義した X に対して，$Y=2^X-1$ となるような Y を求め，ST に代入する．ただし，0≦X≦0.5 でなければならない．
FYL2X	ST に定義した X, ST(1) に定義した Y に対して，$Z=Y\cdot LOG_2X$ となる Z を求め，ST(1) に代入してから，レジスタスタックを POP する．ただし，0<X<∞ かつ −∞<Y<+∞ でなければならない．
FYL2XP1	ST に定義した X, ST(1) に定義した Y に対して，$Z=Y\cdot LOG_2(X+1)$ のような Z を求め，ST(1) に代入してから，レジスタスタックを POP する．ただし，$0\leqq\|X\|<(1-\sqrt{2}/2)$ かつ −∞<Y<+∞ でなければならない．

暗黙的に ST または ST (1) に決められている．表 9・4 には基本的な関数命令を示す．この中で，FPREM 命令は ST に定義した値を，ST (1) の値で割ったときの正確な余りを求めるために使用する．ただし，FPREM 命令は 1 回の減算を実行するだけであるから，余りを求めるためにはステータスレジスタの C_2 が 0 になるまで FPREM 命令を繰り返し実行する必要がある．表 9・5 には三角関数，

表 9・6 80287 定数定義命令

オペコード ニーモニック	動　　作
FLDZ	レジスタスタックに +0.0 を PUSH する.
FLD1	レジスタスタックに +1.0 を PUSH する.
FLDPI	レジスタスタックに円周率 π を PUSH する.
FLD2T	レジスタスタックに LOG_2 10 を PUSH する.
FLDL2E	レジスタスタックに LOG_2 e を PUSH する.
FLDLG2	レジスタスタックに LOG_{10}2 を PUSH する.
FLDLN2	レジスタスタックに LOG_e 2 を PUSH する.

指数関数，対数関数を含む超越関数の命令を示す．しかし，80287 がもつ超越関数の命令は，その適用範囲が限定されたものとなっている．

たとえば，三角関数には，tan と arc tan しかなく，また FPTAN 命令を適用できる範囲は，0 から $\pi/4$ までである．しかし，0 から $\pi/4$ までの tan が求められれば，任意の角度に対する tan はプログラムで計算できる．また，FPTAN 命令の結果は，レジスタスタックの最上位の ST，ST (1) に，Y/X の比の形で求められるから，sin, cos を計算するプログラムも簡単に作成することができる．指数関数，対数関数についても，表 9・5 に示した 3 つの命令だけであるが，80287 は表 9・6 に示すような定数をレジスタスタックに PUSH する命令をもっているので，これらの命令を組み合わせれば，一般的に使用できる指数関数，対数関数のプログラムを作ることができる．

9-5　80286 と 80287 の接続

〔1〕 **80286 との接続**　8087 がホスト CPU として 8086 を必要としたように，80287 はホスト CPU として，80286 または 80386 を使用しなければならない．しかし，80286 と 80287 の間の制御は，8086 と 8087 間の場合とはまったく異なる方式を採用している．図 9·7 に 80286 と 80287 の接続を示す．この図において，80287 は 80286 に対して一種の I/O インタフェースであると考えることができる．表 9·7 に示すように，80287 に対して，0F8H，0FAH，0FCH の 3 つの I/O

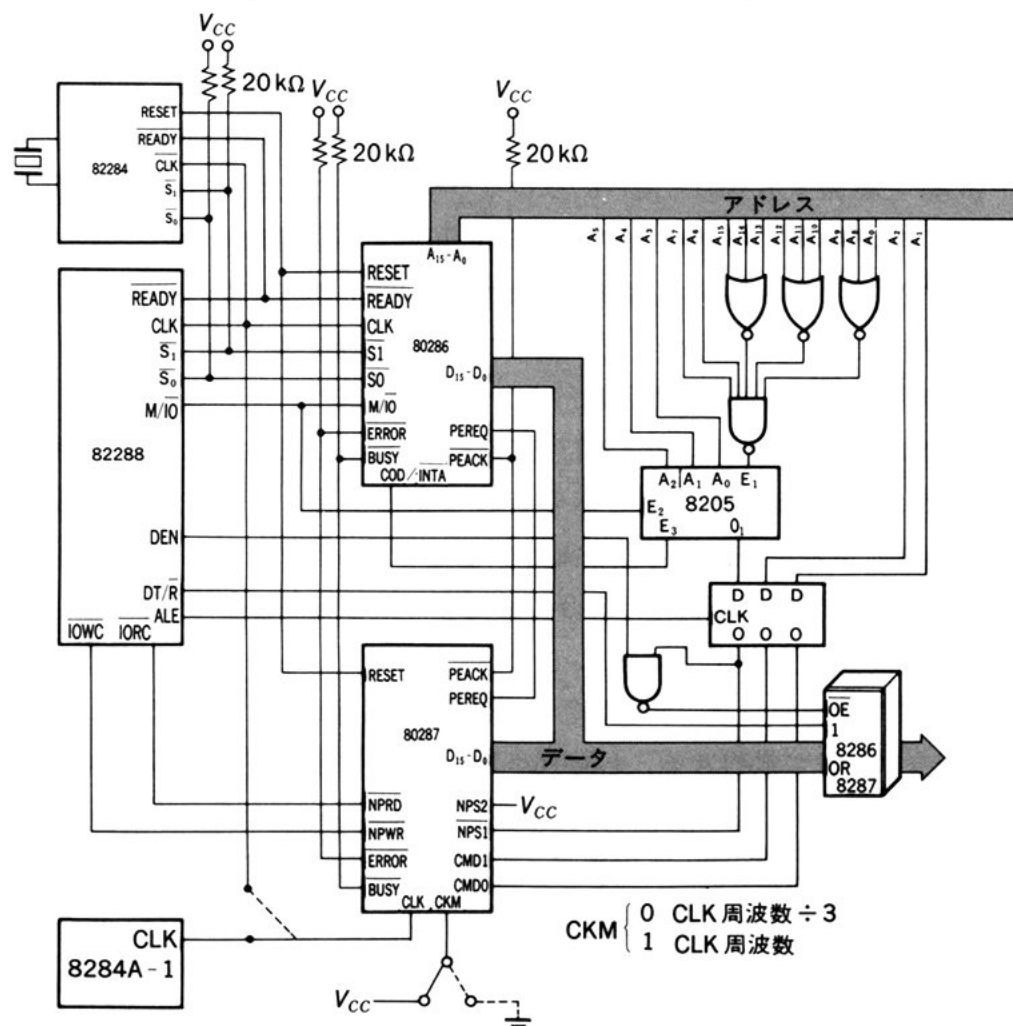



図 9·7　80286 と 80287 の接続

(iAPX286 Programmer's Reference Manual (Order No. 210498-003), D-5, Fig4 より引用)

表9・7 80287のI/Oアドレス

I/Oアドレス	80287のセレクト信号とコマンド入力			
	NPS 2	$\overline{\text{NPS 1}}$	CMD 1	CMD 0
00F8H	1	0	0	0
00FAH	1	0	0	1
00FCH	1	0	1	0
00FEH	I/Oアドレス00FEHは，将来のために予約されている．			

アドレスが与えられている．また，80286は，80287に対して一種のDMAコントローラであると考えることができる．

8087の場合は，8086との間で非常に密な制御を行い，8086と8087のペアを1つのCPUのように見ることができた．しかし，8087に採用された制御方式に欠点がないわけではなかった．それは，8087は8086と同期して動作しなければならず，8086に供給するものと同一のクロックを8087にも供給しなければならなかった．これは8086の製造と8087の製造を同じように進めなければならないことを意味する．8087の場合，8 MHzで動作する8087の製造が遅れて，8 MHzの8086を使用するユーザも，8087を接続して使用するときは5 MHzのクロックで動作させなければならない時期があった．80286は80287をI/Oインタフェースとして見ているために，ホストCPUの動作速度がコプロセサの動作速度に制限されることはない．80287が80286のI/Oの立場になったもう一つの理由は，80286が非常に厳密なメモリ管理を実行するためである．したがって，80287は数値演算だけを実行し，80287の命令におけるメモリ参照も含めて，すべてのメモリ参照を80286が実行する．

〔2〕 **80287のクロック**　80287の内部回路は，8087と同様にデューティ比が33%のクロックで動作する．したがって，82284が発生するシステムクロックを80287の内部回路に供給する場合は，システムクロックを3分周しなければならない．このために，80287は内部にCLK端子から入力するクロックを3分周する分周回路をもつ．80287のCKM端子をLowにした場合，分周回路を介して，内部回路にクロックを送ることができる．また，CKM端子をHighにすれば，分周回路を介さずにCLK端子から入力されたクロックをそのまま内部回路に供給することもできる．図9・7に示す例において，82284が発生する16 MHzのシステムクロックを80287に供給するときは，CKM端子をLowにしなけれ

ばならない．このとき，80287の内部回路は5.3 MHzで動作する．80287は80286とは非同期で動作できるから，システムクロックを80287に供給する必要はない．デューティ比33%のクロックを発生する8284 Aのクロックを，80287のCLK端子に供給することも可能である．このとき，CMK端子をHighに接続する．8284 Aが8 MHzのクロックを供給するならば，80287の内部回路も8 MHzで動作する．

〔3〕 **80287の命令の実行**　図9·7において，80286と80287の動作について考える．80286は，80287の命令（80287の命令を総称して**ESC命令**と呼ぶ）も含めて，すべての命令をフェッチする．もし，フェッチした命令が80286の命令であれば80286内部で処理し，フェッチした命令が80287の命令であれば，80286はその命令コードを，ちょうどI/Oにデータを出力するときのように80287に送る．80287は80286から送られた命令をデコードして，実行することができる．80287での命令の実行は80286の動作とは完全に非同期で進む．したがって，80287が数値演算を実行している間，80286は次の命令を実行しているかもしれない．

しかし，ここで1つの問題がある．それは，80287が前に送られた命令を実行中であるにもかかわらず，80286から新しいESC命令が送られた場合，前の命令の実行が途中でつぶれてしまう．そこで，80287は実行中であるかないかを表す$\overline{\text{BUSY}}$を80286に供給する．80286は$\overline{\text{BUSY}}$がLowであれば，$\overline{\text{BUSY}}$が再びHighになるまで，ESC命令のコードを80287に送らない．このとき，80286は命令コードのデコードを一時的に休むことになる（ただし，ESC命令の中には$\overline{\text{BUSY}}$とは無関係に実行されるものもある）．

なお，80286の$\overline{\text{BUSY}}$端子は，80287とのソフトウェアの同期をとる以外の目的に使用してもかまわない．このために，80286はWAIT命令をもつ．80286がWAIT命令を実行したとき，$\overline{\text{BUSY}}$がLowであれば，$\overline{\text{BUSY}}$が再びHighになるまで次の命令を実行しない．また，WAIT命令は，図9·8に示すように80286の命令が80287の命令完了を待たなければならない場合にも使用される．

ESC命令の中で，FLD命令またはFST命令のように，メモリからデータをリードしたり，メモリにデータをライトするものがある．このとき，80287は80286をDMAコントローラとして使用する．このときの制御はPEREQと$\overline{\text{PEACK}}$によって実行される．80287がメモリからデータをリードするときは，80286が

図 9・8　WAIT 命令による 80286 と 80287 の同期

メモリからデータをリードしてから，80287 にライトする．また，80287 がメモリにデータをライトするとき，80287 からの要求に従って，80286 が 80287 からリードしたデータをメモリにライトする．

8087 の WAIT 命令

8087 は 8086 と同じタイミングで命令をフェッチしてから, また 80287 は 80286 から命令コードを供給されてから命令のデコードを開始し, その処理を実行する. どちらの場合にも現在実行中の命令があるにもかかわらず，次の命令をデコードした場合, 前の命令の処理は途中で壊れ結果は残らない．このため, 8087 も 80287 も $\overline{\mathrm{BUSY}}$ 端子に実行中か実行中でないかを表す記号を出力し，ホスト CPU において，8086 の場合は 8087 の実行が終了するまで次の命令のデコードを遅延させる処理が必要となる．80286 の場合は 80287 の実行が終了するまで，80287 に命令コードを供給するのを遅延させる必要がある．

このため 8086 の場合は，ESC命令の前に必ず WAIT命令を挿入する．しかし 80286 の場合，ESC命令自体の中に WAIT命令と同じ機能をもつため，わざわざ WAIT命令を ESC命令の前に入れる必要はない．

9-6 例外処理

図9·9に示すように，80287は演算処理において6種類のエラーを検査する．もし，エラーが発生した場合，80287はステータスレジスタのビット0からビット5までのエラーフラグの対応するビットを1にする．さらに，コントロールレジスタのビット0からビット5までの対応するマスクビットが0であれば，ステータスレジスタのESを1にして，$\overline{\text{ERROR}}$端子にLowの信号を出力する．

8087の場合は，割り込みコントローラ8259Aを介して，8086に割り込み信号を供給したが，80287の場合は，80287の$\overline{\text{ERROR}}$出力端子を直接80286の$\overline{\text{ERROR}}$入力端子に接続すればよい．$\overline{\text{ERROR}}$がLowのときに，80286がESC命令またはWAIT命令を実行すると，タイプ16の割り込みを発生する．

タイプ16の割り込み処理で80287の例外処理を実行することができる．ステータスレジスタのESが1である間，80287は$\overline{\text{ERROR}}$端子にLowの信号を出力し続けるので，例外処理においてFNCLEX命令を実行して，ESを0にする必要がある．FNCLEX命令はステータスレジスタのビット0からビット5までのエラーフラグ，ES，そしてビット15のBを0にクリアする命令であり，FNCLEX命令自体は$\overline{\text{ERROR}}$の状態を無視して実行される．このように，80287の制御命令の中には$\overline{\text{ERROR}}$の状態に関係なく実行されるものがある．

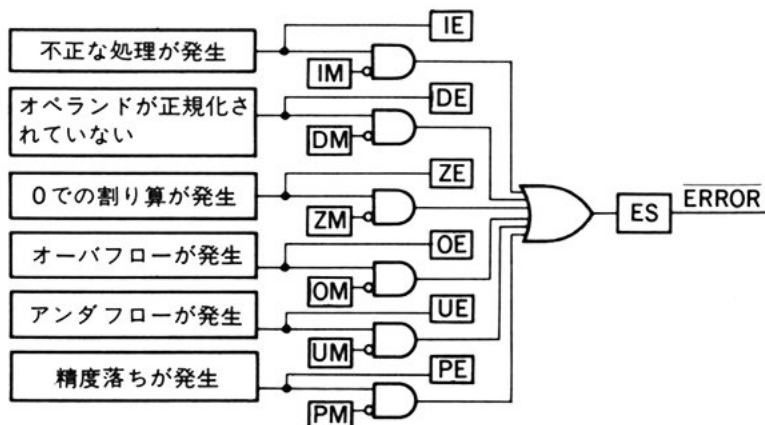



図 9・9　80287の$\overline{\text{ERROR}}$出力

9-7　80287 のためのサポート

〔1〕 **80286 のサポート**　80287 は 80286 から命令コードを与えられて初めて動作する．80287 自体がメモリなどを直接参照することはなく，80287 外部への参照はすべて 80286 が実行する．80287 に対して 80286 が実行する動作は図 9·10 に示すように，MSW の MP，EM，TS の 3 ビットのフラグによって制御することが可能である．

80287 がハードウェアに存在するとき，MP を 1 に設定し，EM を 0 に設定する．逆にハードウェアに 80287 がなく，80287 と同じ機能をもつソフトウェアエミュレータによって 80287 を代用したいときには，MP を 0 に設定し，EM を 1 に設定する．TS については，80286 がタスクスイッチを発生したとき自動的に 1 を設定する．

〔2〕 **タスクスイッチにおける 80287**　タスクスイッチによって，80286 のレジスタの状態はすべて TSS に自動的に保存されるが，80287 のレジスタについては，プログラムの実行によって保存しなければならない．このとき，TS を利用することができる．すなわち，MP=1，EM=0，TS=1 の条件において ESC 命令または WAIT 命令を実行したいとき，タイプ 7 の割り込みが発生する．TS=1 はタスクスイッチが発生したことを意味するから，いま実行しようとした ESC 命令または WAIT 命令は別のタスクの命令である．したがって，これらの命令を実行する前に，80287 のすべてのレジスタの状態をメモリに保存し，かつ，80287 を初期化してから逆に新しいタスクで使用するレジスタの状態をメモリから80287

MSW		TS	EM	MP	PE

MP { 80287 が存在するとき 1 に設定する．
80287 が存在しないとき 0 に設定する．

EM { 80287 命令の実行をソフトウェアエミュレータで代用するとき 1 に設定する．
80287 命令を 80287 で処理するとき 0 を設定する．

TS { タスクスイッチが発生したとき 1 になる．
80286 をリセットしたときまたは CLTS 命令を実行した後，0 になる．

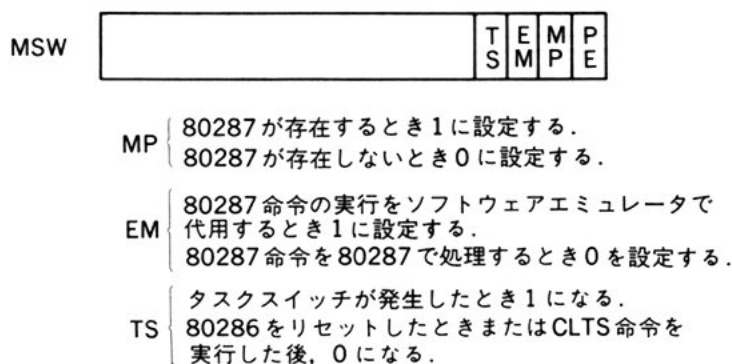



図 9·10　80286 側のサポート

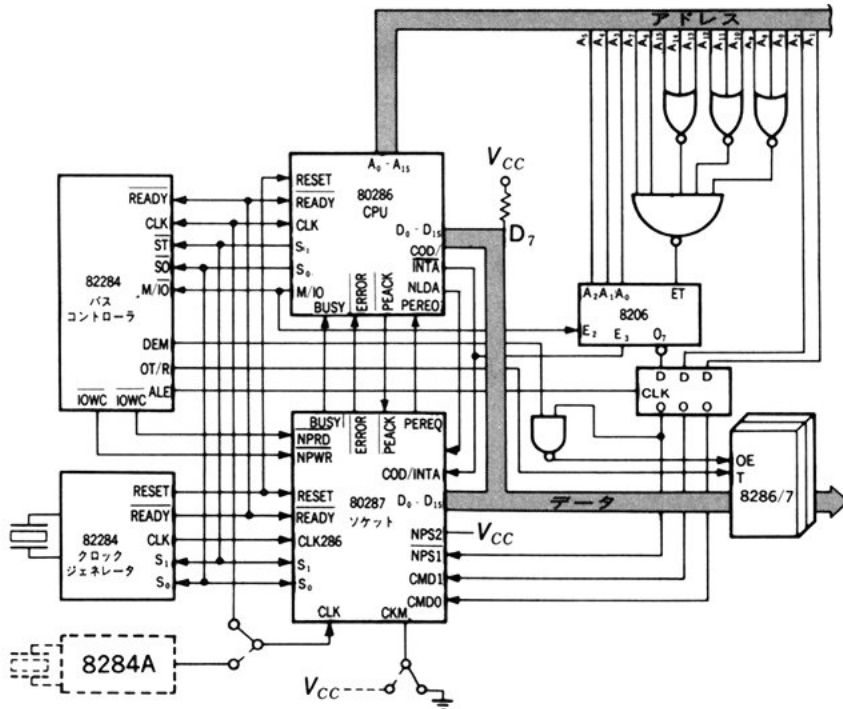



図 9・11　80287の自動検査
(iAPX286 Hardware Reference Manual (Order No. 210760-001), 6-2, Fig 6-1 より引用)

```
FND_287:FNINIT            ➡80287 内部状態を初期化
        FSTSW AX          ➡AX ← SW
        OR AL,AL          ➡AL=0 であるかどうかを調べる
        JZ GOT_287        ➡AL=0 のとき GOT_287 へジャンプする
        SMSW AX           ➡80287 をもたない場合
        OR AX,04H         ➡EM ← 1
        LMSW AX
        JMP CONTINUE
GOT_287:SMSW AX           ➡80287 をもつ場合
        OR AX,02H           MP← 1
        LMSW AX
CONTINUE:                 ➡処理を続ける
```

図 9・12　80287 自動検査プログラム
(iAPX286 Hardware Reference Manual (Order No. 210760-001), 6-8, Fig 6-4 より引用)

に代入する必要がある．80286 の TSS はオフセット 0 からオフセット 43 までが定義されており，特に 80287 のレジスタ状態を保存するための領域は定義されていないが，TSS のリミットを変更し，オフセット 44 からの領域を使用すればよい．

タイプ 7 の割り込みによって，以上のような処理を実行してから，IRET 命令を実行すれば，割り込みの原因となった ESC 命令または WAIT 命令を新しいタスクにおいて再実行することができる．なお，後の処理において再びタスクスイッチを判断するため，割り込み処理の中で CLTS 命令を使用して，TS を 0 に戻しておく必要がある．

〔3〕 80287 エミュレータ　もし，浮動小数点演算を高速で実行する必要がなければ，80287 を使用せずに，同等の機能をもつ手続きを利用することによって浮動小数点演算を実現してもかまわない．このとき，EM=1，MP=0 にした状態で ESC 命令を実行すれば，タイプ 7 の割り込みを発生するので，割り込み処理によって 80287 のエミュレータを実現することができる．

〔4〕 MSW の初期設定　以上のように 80287 を使用するか，しないかによって MSW の MP，EM の 2 ビットを初期設定しなければならない．図 9･11 に 80287 の存在を検査する 1 つの方法と，図 9･12 に MSW の MP, EM を初期設定するプログラムを示す．すなわち，図 9･11 に示したように，80286 と 80287 を接続するデータバスの D_7 を抵抗でプルアップしておき，図 9･12 に示したプログラムで，80287 のレジスタの状態を初期化する命令 FNINIT を実行してから，FSTSW 命令を使用して 80287 のステータスワードの値を AX に代入する．初期化後の 80287 のステータスレジスタの下位 8 ビットは 0 であるから，もし 80287 が存在すれば，AL は 0 である．これに対して，80287 が存在しなければ，少なくともデータバスの D_7 はプルアップされているので，AL には必ず 0 以外の値が代入される．したがって，AL の値が 0 であるかどうかを判定することによって，80287 の状態に応じた MSW の初期設定を実現することができる．

10. プログラム開発

インテル社が提供するシステム開発環境を知ることは，80286 のプロテクトモードで実行されるシステムを開発するうえで有益である．プロテクトモードでは，GDT, LDT, IDT, TSS などの各種のテーブルを定義し，またセグメントディスクリプタ，ゲートなどの各種のディスクリプタをテーブルに定義しなければならない．さらに，プログラム中で参照するセレクタが間違いなく，対応するディスクリプタを指定しなければならない．もちろん，ディスクリプタには間違いのない値を初期設定しなければならない．アセンブラ，リンカ程度のツールだけでこれだけの作業を行うためには，超人的な注意力が必要である．80286 のプロテクトモードの特性を十分に生かしたシステムを開発することは，すなわちマルチタスクで動作するリアルタイム OS を設計することになるのである．しかし，8086 の時代におけるセグメンテーションの問題についても同様だが，このような扱いのうえでの困難さは高機能のマクロアセンブラ，高機能のユーティリティプログラムによって十分解決できる．

10-1 開発システム

現在 80286 の開発システムとしては，図 10・1 に示すインテル社の MDS シリーズ Ⅳ などがある．プログラム言語には，80286 のアセンブリ言語 ASM 286 の他に，PL/M-286，PASCAL-286，FORTRAN-286 などの高級言語が用意されている．また，MDS シリーズ Ⅳ は RS-232 C ラインで ROM ライタを接続でき，プログラムの ROM 化までの開発をスムースに行うためのソフトウェアが用意されている．シリーズ Ⅳ 自体は 8086 で動作するシステムであるが，80286 のシミュレータ SIM 286 のようなソフトウェアも用意されており，簡単なソフトウェアデバッグであればシリーズ Ⅳ で実行することができる．

シリーズⅣ以外では，パーソナルコンピュータで **UDI** と呼ばれる一種のソフトウェアインタフェースを介して，ASM 286 などのインテルのソフトウェアを使用することも可能である．

また，ハードウェア，ソフトウェアをゼロの状態から開発するのではなく，図 10・2 に示すシステム 310 のような箱入りの 80286 コンピュータを利用することもできる．システム 310 では，XENIX 286 またはリアルタイム OS である iRMX 286 が用意されており，ユーザは応用プログラムを書くだけでよい．

図 10・1　MDS シリーズ Ⅳ

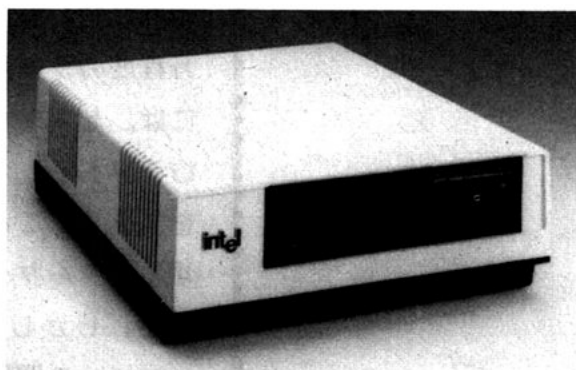



図 10・2　システム 310

10-2 システム開発とユーティリティプログラム

〔1〕 **プログラム開発環境**　ASM 286 などのトランスレータを使用して，ソースプログラムを再配置形式オブジェクトプログラムに変換した後，**BND 286**（バインダと呼ぶ），**BLD 286**（ビルダと呼ぶ）などのユーティリティプログラムを使用して，実行可能なプログラムを作成する．ここで，バインダは別々のファイルに作成したオブジェクトプログラムを1つに結合するプログラムで，リンカに相当するユーティリティプログラムである．また，ビルダはセグメントの絶対アドレスを定義するいわゆるロケータに相当するユーティリティプログラムである．ただし，ビルダの働きはロケータの機能だけではない．プログラマはシステムで使用するセグメントディスクリプタ，各種のゲート，TSS，GDT，LDT，IDT などの定義を**ビルダプログラム**と呼ばれる書式で，PL/M などのプログラムのソースファイルを作成するのと同様に，ファイルに作っておけば，後はビルダ BLD 286 が

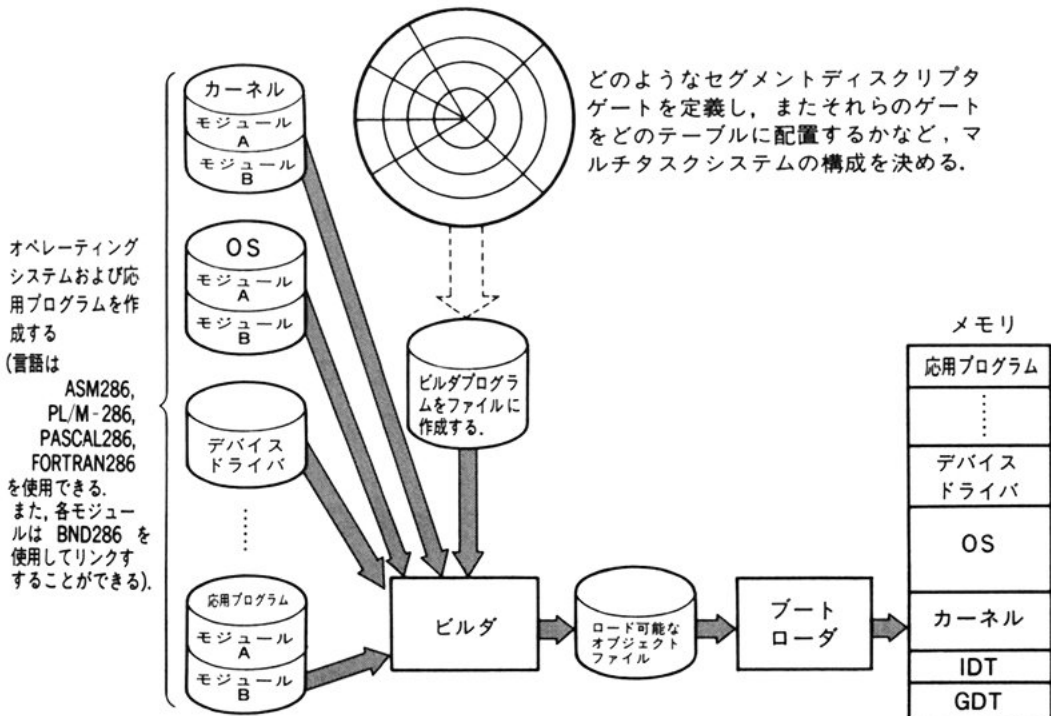



図 10・3　静的システムの開発
(iPX286 Operating Systems Writer's Guide (Order No. 121960-001), 1-10, Fig 1-8 より引用)

ビルダプログラムをコンパイルしてから，ディスクリプタなどのデータをユーザプログラムにリンクする．このとき，ビルダはビルダプログラムの中に明らかな定義の誤りを発見すればエラーメッセージを出力し，またユーザプログラムの定義とビルダプログラムの定義の間に不合理と思われるような点を発見したときは注意メッセージを出力する．このようにディスクリプタ，ゲートなどの 80286 のプロテクトモードシステムに必要なデータをビルダプログラムでシンボリックに定義できるため，かなりのエラーを BLD 286 を実行する段階で排除することができる．

〔2〕 **システム開発の流れ**　次に，典型的な 2 つの例についてシステム開発の流れを考える．

（a） **静的システムの開発**　図 10·3 に静的システムの開発の流れを示す．ここでいう**静的システム**とは図に示すように，システムの電源を投入した後，ブートローダと呼ばれるプログラムがカーネル，デバイスドライバなどの OS をリンクした応用プログラムをディスクファイルからメモリに転送して実行するようなシステムである．ブートローダは ROM メモリに記録されていると考えればよい．このようなシステムでは実行の間 OS も応用プログラムもメモリに常駐する．FA（ファクトリオートメーション）などにおける装置の制御システムにこのような形態が多いだろう．

ブートローダによって読まれるオブジェクトファイルにはセグメント，ディスクリプタテーブルなどの配置アドレスが定義されている．このような絶対番地形式のファイルを作成するユーティリティプログラムがビルダである．ビルダは OS のカーネル，デバイスドライバなどの手続きと応用プログラム，さらにビルダプログラムで定義した GDT，IDT，TSS などをリンクしてから絶対番地形式のオブジェクトファイルを作る．

（b） **動的システムの開発**　動的システムの開発の流れを図 10·4 に示す．**動的システム**とは最初 OS がブートローダによってメモリに転送された後，メモリに常駐する OS の管理の下でさまざまな応用プログラムが実行されるようなシステムである．このシステムでは OS がメモリの空き領域を管理し，応用プログラム自体は再配置形式で作成しておき，プログラムを配置するメモリのアドレスを OS のローダが決める．このような 1 つの応用プログラムを定義した再配置形式のオブジェクトファイルを作るためにバインダ BND 286 を使用する．バインダはい

くつかの再配置形式のオブジェクトファイルをリンクして，実行可能なオブジェクトファイルを作成する．このときビルダが作った**エクスポートファイル**もリンクすることができる．ビルダは作成するプログラムの中で使用されているゲートのセレクタ，変数のセレクタとオフセットなどの情報だけをエクスポートファイルと呼ばれるファイルを出力することができる．ユーザは，このエクスポートファイルをリンクすることによって，OS 内で定義された手続きを引用するようなプログラムを作成することができる．

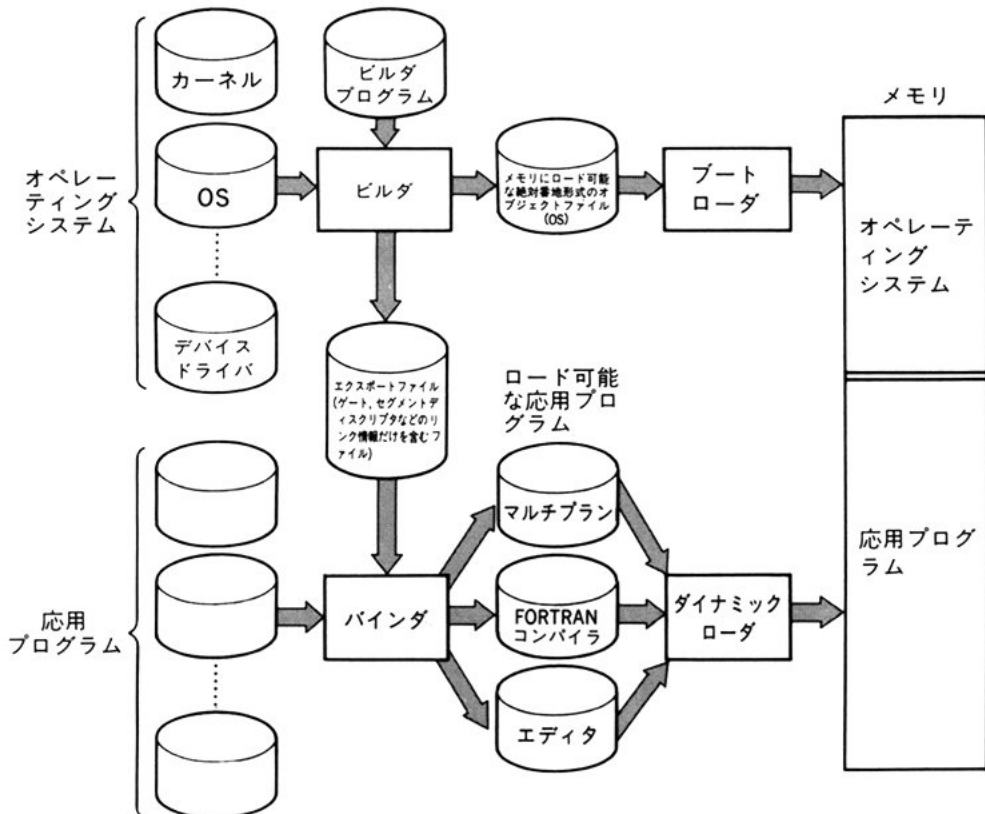



図 10・4　動的システムの開発
(iPX286 Operating Systems Writer's Guide (Order No. 121960-001), 1-11, Fig 1-9 より引用)

10-3 デバッグ

〔1〕 **デバッグツール**　一般にマルチタスクシステムのデバッグは困難である．マルチタスクシステムに限らずプログラムデバッグには王道というものはなく，結局は**モジュール化による階層的なシステム開発**が最良の方法であるように思う．しかし，そのうえで使いやすいデバッグツールを使用することは有効である．デバッグ用の道具にはSIM 286のようなソフトウェアの論理エラーをデバッグするソフトウェアデバッガもあるが，マイクロコンピュータのハードウェアおよびソフトウェアの開発には**ICE**(一般にアイスと読まれる)と呼ばれるデバッグツールが利用される．

図10・5にインテル社が提供するI²ICE（アイスクェアアイスと読まれる）を示す．I²ICEは，パーソナリティモジュールと呼ばれる部分を交換することによって8086，80186，80286のデバッグをすることができる．ICEは**インサーキットエミュレータ**を意味し，名前のようにハードウェアシステムのエミュレーションを行う機能をもつ．

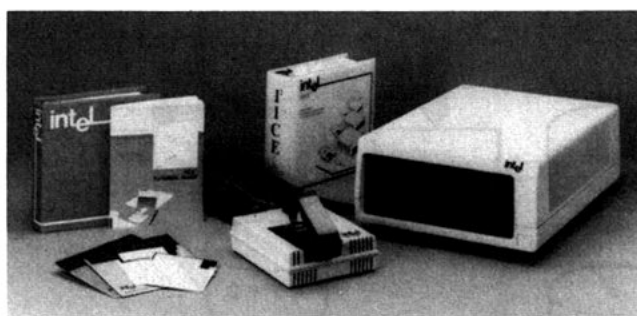



図10・5　I²ICE

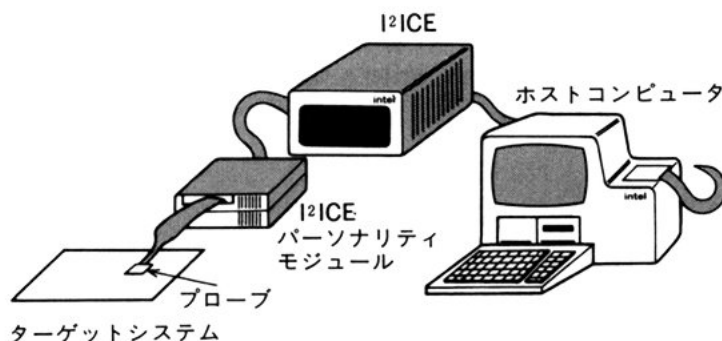



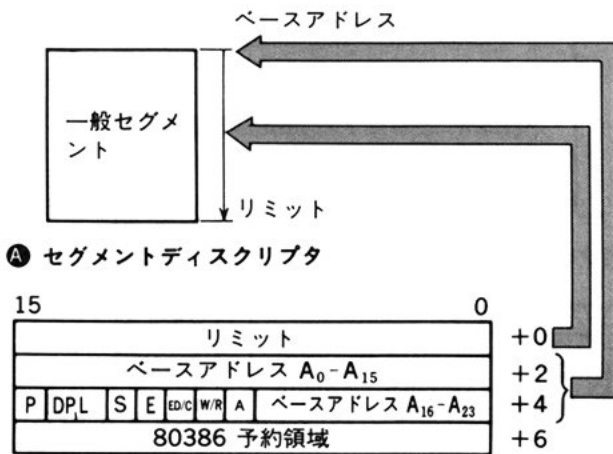
図10・6　I²ICEの使用

たとえば，I²ICE の場合は図 10･6 に示すように，I²ICE のプローブをターゲットシステム（最終的にソフトウェアを実行するハードウェアシステム）の 80286 のソケットに差し込み，ホストコンピュータからターゲットシステムのメモリにプログラムを配置して実行することができる．また，ターゲットシステムのハードウェアが未完成の状態であっても，I²ICE のもつメモリまたはホストコンピュータのもつメモリを代用して，プログラムのデバッグを行うことができる．

〔2〕 **プロテクトモードのデバッグ**　80286 はセグメントキャッシュの値による厳密なメモリ管理を実行する．さらにプロテクトモードにおいてはセグメントキャッシュと GDT, LDT に定義されるセグメントディスクリプタの関係が 1 対 1 に対応し，80286 はディスクリプタテーブルの値によるメモリ管理または TSS によるタスク管理を実行する．しかし，プログラムのデバッグを実行するときには，これらの保護のための壁をすべて透明にして，80286 のすべての状態をモニタしたい場合がある．そのために，I²ICE がもつ 80286 は一般の 80286 とは異なり，保護機能をはずせるようになっている．したがって，PCHECK という I²ICE の内部スイッチ変数を OFF にすれば，I²ICE の 80286 はデバッグの対象となるプログラムには保護を実行するが，ホストコンピュータの前でデバッグ作業をするオペレータは，本来は見ることができないキャッシュの中の状態をも CRT に表示してモニタすることができる．さらに，キャッシュの中のベースアドレス，リミット，アクセスライトの値を直接に書き換えることさえできるのである．

付　　　録

I. セグメントおよびディスクリプタのまとめ

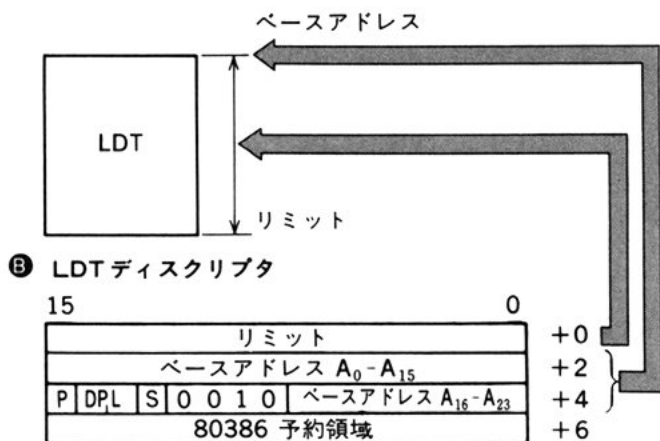
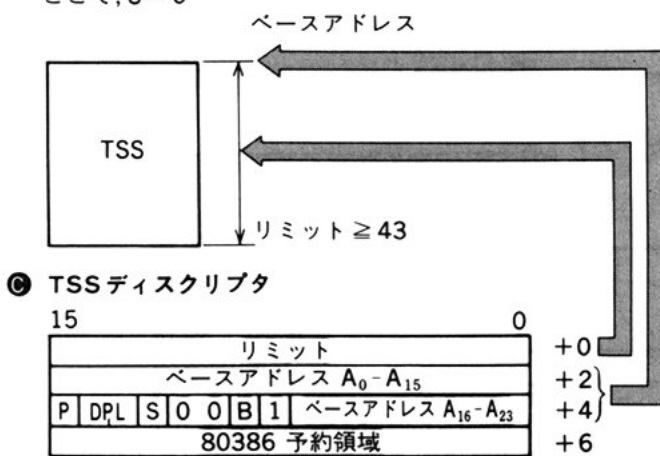
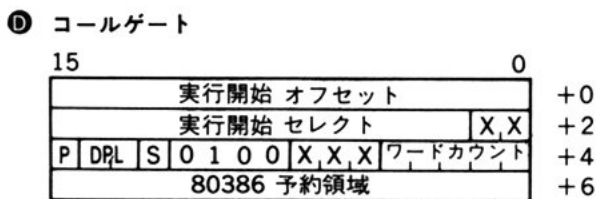
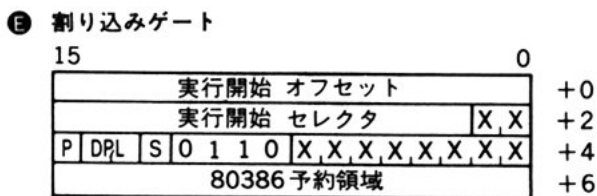
図は，80286 システムで定義されるセグメントおよびディスクリプタの種類をまとめたものである．

ここで，S＝1

セグメントディスクリプタの種類は E, ED/C, W/R の3ビットによって決まる

E＝0のとき	実行不可，リード可
このとき	
ED＝0	0≦オフセット≦リミットの領域だけ使用可
ED＝1	リミット＋1≦オフセット≦0FFFFHの領域だけ使用可
W＝0	ライト不可
W＝1	ライト可
E＝1のとき	実行可，ライト不可
このとき	
C＝0	ノンコンフォーミングコードセグメント (一般特権規則を適用)
C＝1	コンフォーミングコードセグメント (例外特権規則を適用)
R＝0	リード不可
R＝1	リード可

ベースアドレス
LDT
リミット
Ⓑ LDTディスクリプタ
15
0
リミット
ベースアドレス A_0-A_{15}
P
DPL
S
0 0 1 0
ベースアドレス A_{16}-A_{23}
80386 予約領域
+0
+2
+4
+6
ここで，S＝0
ベースアドレス
TSS
リミット≧43
Ⓒ TSSディスクリプタ
15
0
リミット
ベースアドレス A_0-A_{15}
P
DPL
S
0 0 B 1
ベースアドレス A_{16}-A_{23}
80386 予約領域
+0
+2
+4
+6
ここで，S＝0

Ⓓ コールゲート
15
0
実行開始 オフセット
実行開始 セレクト
X X
P
DPL
S
0 1 0 0
X X X
ワードカウント
80386 予約領域
+0
+2
+4
+6
ここで，S＝0

Ⓔ 割り込みゲート
15
0
実行開始 オフセット
実行開始 セレクタ
X X
P
DPL
S
0 1 1 0
X X X X X X X X
80386予約領域
+0
+2
+4
+6
ここで，S＝0

Ⓖ タスクゲート

15							0	
X X X X X X X X X X X X X X X X								+0
TSSセレクタ								+2
P	DPL	S	0 1 0 1	X X X X X X X X				+4
80386 予約領域								+6

ここで，S＝0

Ⓕ トラップゲート

15						0	
実行開始 オフセット							+0
実行開始 セレクタ					X X		+2
P	DPL	S	0 1 1 1	X X X X X X X X			+4
80386 予約領域							+6

ここで，S＝0

各ディスクリプタを定義できるディスクリプタテーブルの関係

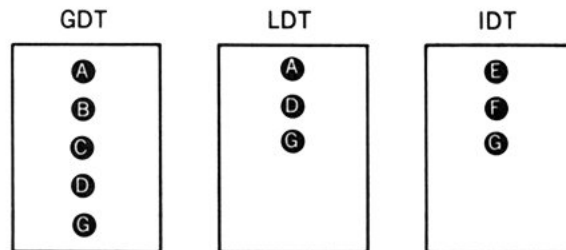



図　ディスクリプタの種類

II. 80286 命令コード

表付·1 に 80286 の命令コードを示す．また，命令コードの中の mod, disp, r/m, reg の識別を表付·2 から表付·5 に示す．80286 が実行する重要な保護動作，タスクスイッチの動作などの処理の流れをリスト (a) から (h) までに示す．このリストにおいて，DT は GDT または LDT のディスクリプタテーブルを表し，DT (セレクタ) によってセレクタによって識別されるテーブル内の特定のディスプリクタを表すものとする．さらに，ディスクリプタ内の P, DPL, E, C, W, LIMIT などの各フィールドは DT (セレクタ). P, DT (セレクタ). DPL, DT (セレクタ). LIMIT のように表している．また，DTR は GDTR または LDTR のどちらかのディスクリプタテーブルレジスタを表す．GDTR か LDTR かの区別は，その状況で扱っているセレクタの T I ビットによって区別できる．

また，80286 が発生する保護エラーは

タイプ 13 … GP error
タイプ 11 … NP error
タイプ 12 … SS error
タイプ 10 … TS error

のように表現している．さらに続く括弧の中にスタックに PUSH されるエラーコードを指定する．

表 付·1　80286 命令コード

命令ニーモニックとコード	コメント	リアルモード	プロテクトモード
MOV (move)		○	○
[1 000100 w] [mod reg r/m]	reg/mem ← reg		
[1 000101 w] [mod reg r/m]	reg ← reg/mem		
[1 100011 w] [mod 0 0 0 r/m] [data] [data if w=1]	reg/mem ← imm		
[1 011w reg] [data] [data if w=1]	reg ← imm		
[1 010000 w] [addr-low] [addr-high]	accum ← mem		
[1 010001 w] [addr-low] [addr-high]	mem ← accum		
[1 000111 0] [mod 0 reg r/m]	sreg ← reg/mem		リスト (a), (b) 参照
[1 000110 0] [mod 0 reg r/m]	reg/mem ← sreg		
PUSH (push)		○	○
[1 111111 1] [mod 110 r/m]	push mem		
[0 1010 reg]	push reg		
[0 00reg11 0]	push sreg		
[0 11010s 0] [data] [data if s=0]	push imm		
PUSHA (push all)		○	○
[0 110000 0]	すべてのワード汎用レジスタをスタックに PUSH する		

命令ニーモニックとコード	コメント	リアルモード	プロテクトモード
POP (pop)		○	○
1 000111 1 | mod 000 r/m	pop mem		
0 1011 reg	pop reg		
0 00reg11 1 (reg ≠ 01)	pop sreg		リスト (a) 参照
POPA (pop all)		○	○
0 110000 1	スタックからすべてのワード汎用レジスタをPOPする.		
XCHG (exchange)		○	○
1 000011 w | mod reg r/m	reg/mem ⇄ reg		
1 0010 reg	reg ⇄ accum		
IN (in)		○	○
1 110010 w | port	accum ← port で指定した I/O		
1 110110 w	accum ← DX で指定した I/O		
OUT (out)		○	○
1 110011 w | port	port で指定した I/O ← accum		
1 110111 w	DX で指定した I/O ← accum		
XLAT (translate byte to AL)		○	○
1 101011 1	オフセット＝(BX+AL)のメモリの1バイトの値をALに代入する.		
LEA (lood effective address to register)		○	○
1 000110 1 | mod reg r/m	reg で指定したレジスタに r/m で指定したメモリのオフセットを代入する.		
LDS (load pointer to DS : reg)		○	○
1 100010 1 | mod reg r/m (mod ≠ 11)	r/m で指定したメモリの上位ワードの値を DS に代入し，下位ワードを reg で指定したレジスタに代入する.		リスト (a) 参照
LES (load pointer to ES : reg)		○	○
1 100010 0 | mod reg r/m (mod ≠ 11)	r/m で指定したメモリの上位ワードの値を ES に代入し，下位ワードを reg で指定したレジスタに代入する.		リスト (a) 参照
LAHF (load AH with flags)		○	○
1 001111 1	AH ← フラグレジスタの下位バイト		
SAHF (store AH into flags)		○	○
1 001111 0	フラグレジスタの下位バイト ← AH		
PUSHF (push flags)		○	○
1 001110 0	フラグレジスタの値をスタックに PUSH する.		
POPF (pop flags)		○	○
1 001110 1	スタックから POP した値をフラグレジスタに代入する.		

命令ニーモニックとコード	コメント	リアルモード	プロテクトモード
ADD (add) 0 00000d w mod reg r/m 1 00000s w mod 000 r/m data data if s w=01 0 000010 w data data if w=1	(d=1) reg←reg+reg/mem (d=0) reg/mem←reg/mem +reg reg/mem←reg/mem +imm accum←acum+imm	○	○
ADC (add with carry) 0 00100d w mod reg r/m 1 00000s w mod 010 r/m data data if s w=01 0 001010 w data data if w=1	(d=1) reg←reg+reg/mem +carry flag (d=0) reg/mem←reg/mem +reg+curry flag reg/mem←reg/mem +imm+carry flag accum←accum+imm +carry flag	○	○
INC (increment) 1 111111 w mod 000 r/m 0 1000 reg	reg/mem←reg/mem+1 reg←reg+1	○	○
SUB (subtract) 0 01010d w mod reg r/m 1 00000s w mod 101 r/m data data if s w=01 0 010110 w data data if w=1	(d=1) reg←reg−reg/mem (d=0) reg/mem←reg/mem −reg reg/mem←reg/mem −imm accum←accum−imm	○	○
SBB (subtract with borrow) 0 00110d w mod reg r/m 1 00000s w mod 011 r/m data data if s w=01 0 001110 w data data if w=1	(d=1) reg−reg/mem −carry flag (d=0) reg/mem←reg/mem −reg−carry flag reg/mem←reg/mem −imm−carry flag accum←accum −imm−carry flag	○	○
DEC (decrement) 1 111111 w mod 001 r/m 0 1001 reg	reg/mem←reg/mem−1 reg←reg−1	○	○
CMP (compare) 0 011101 w mod reg r/m 0 011100 w mod reg r/m 1 00000s w mod 111 r/m data data if s w=01 0 011110 w data data if w=1	reg−reg/mem reg/mem−reg reg/mem−imm accum−imm	○	○
NEG (negate) 1 111011 w mod 011 r/m	2の補数をとる．	○	○
AAA (ASCII adjust for add) 0 011011 1	AL にある純２進数をアンパック ト BCD に補正する．純２進加算を実行した後に使用する．	○	○

命令ニーモニックとコード	コメント	リアルモード	プロテクトモード
DAA (decimal adjust for add) | 0 010011 1 |	ALにある純2進をパックトBCDに補正する. 純2進加算を実行した後に使用する.	○	○
AAS (ASCII adjust for subtract) | 0 011111 1 |	ALにある純2進をアンパックトBCDに補正する. 純2進減算を実行した後に使用する.	○	○
DAS (decimal adjust for subtract) | 0 010111 1 |	ALにある純2進をパックトBCDに補正する. 純2進減算を実行した後に使用する.	○	○
MUL (unsigned multiply) | 1 111011 w | mod 100 r/m |	(w=0) AX←AL＊reg/mem (w=1) DX, AX←AX ＊reg/mem	○	○
IMUL (integer multiply) | 1 111011 w | mod 101 r/m |	(w=0) AX←AL＊reg/mem (w=1) DX, AX←AX ＊reg/mem	○	○
IMUL (integer immediate multiply) | 0 11010s 1 | mod reg r/m | data | data if s=0 |	reg←reg/mem＊imm	○	○
DIV (unsigned divide) | 1 111011 w | mod 110 r/m |	(w=0) { AL←AX ÷reg/mem; AH←AX MOD reg/mem } (w=1) { AX←DX, AX ÷reg/mem; DX←DX, AX MOD reg/mem }	○	○
IDIV (integer divide) | 1 111011 w | mod 111 r/m |	(w=0) { AL←AX ÷reg/mem; AH←AX MOD reg/mem } (w=1) { AX←DX, AX ÷reg/mem; DX←DX, AX MOD reg/mem }	○	○
AAM (ASCII adjust for multiply) | 1 101010 0 | 0 000101 0 |	アンパックトBCDの乗算のための補正命令. 乗算命令を実行してから使用する.	○	○
AAD (ASCII adjust for divide) | 1 101010 1 | 0 000101 0 |	アンパックトBCDの除算のための補正命令. 除算命令を実行する前に使用する.	○	○
CBW (convert byte to word) | 1 001100 0 |	ALにある符号付き整数をAXに拡張する.	○	○

命令ニーモニックとコード	コ　メ　ン　ト	リアルモード	プロテクトモード
CWD (convert word to double word) \[1 001100 1\]	AX にある符号付き整数を DX, AX に拡張する.	○	○
shift/rotate \[1 101000 w | mod TTT r/m\] \[1 101001 w | mod TTT r/m\] \[1 100000 w | mod TTT r/m | count\] TTT 命令ニーモニック 0 0 0　ROL 0 0 1　ROR 0 1 0　RCL 0 1 1　RCR 1 0 0　SHL/SAL 1 0 1　SHR 1 1 1　SAR	reg/mem を 1 ビットだけシフト/回転する. reg/mem を CL ビットだけシフト/回転する. reg/mem を imm ビットだけシフト/回転する.	○	○
AND (and) \[0 01000d w | mod reg r/m\] \[1 000000 w | mod 100 r/m | data | data if w=1\] \[0 010010 w | data | data if w=1\]	(d=1) reg←reg AND reg/mem (d=0) reg/mem←reg/mem AND reg reg/mem←reg/mem AND imm accum←accum AND imm	○	○
TEST (test) \[1 000010 w | mod reg r/m\] \[1 111011 w | mod 000 r/m | data | data if w=1\] \[1 010100 w | data | data if w=1\]	reg/mem AND reg reg/mem AND imm accum AND imm	○	○
OR (or) \[0 00010d w | mod reg r/m\] \[1 000000 w | mod 001 r/m | data | data if w=1\] \[0 000110 w | data | data if w=1\]	(d=1) reg←reg OR reg/mem (d=0) reg/mem←reg/mem OR reg reg←reg OR imm accum←accum OR imm	○	○
XOR (exclusive or) \[0 01100d w | mod reg r/m\] \[1 000000 w | mod 110 r/m | data | data if w=1\] \[0 011010 w | data | data if w=1\]	(d=1) reg←reg XOR reg/mem (d=0) reg/mem←reg/mem XOR reg reg/mem←reg/mem XOR imm accum←accum XOR imm	○	○
NOT (not) \[1 111011 w | mod 010 r/m\]	NOT reg/mem	○	○

命令ニーモニックとコード	コメント	リアルモード	プロテクトモード
MOVS (move string)	ストリング命令	○	○
1 010010 w			
CMPS (compare string)			
1 010011 w			
SCAS (scan string)			
1 010111 w			
LODS (lood string)			
1 010110 w			
STOS (store string)			
1 010101 w			
INS (in string)			
0 110110 w			
OUTS (out string)			
0 110111 w			
REP (repeat prefix)		○	○
1 111001 0	REP の直後にあるストリング命令を CX 回だけ繰り返す.		
REPE/REPZ			
1 111001 1	REPE または REPZ の直後にあるストリング命令を CX 回だけ, または ZF=0 になるまで繰り返す.		
REPNE/REPNZ			
1 111001 0	REPNE または REPNZ の直後にあるストリング命令を CX 回だけ, または ZF=1 になるまで繰り返す.		
CALL (call)		○	○
1 110100 0 | disp-low | disp-high	直接 near call		
1 111111 1 | mod 010 r/m	reg/mem 間接 near call		
1 001101 0 | segment offset | segment selector	直接 far call		リスト (d) 参照
1 111111 1 | mod 011 r/m (mod≠11)	間接 far call		リスト (d) 参照
RET (return)		○	○
1 100001 1	near return		
1 100001 0 | data-low | data-high	near return (SP←SP+imm を伴う)		
1 100101 1	far return		リスト (e) 参照
1 100101 0 | data-low | data-high	far return (SP←SP+imm を伴う)		
JMP (jump)		○	○
1 110101 1 | disp-low	short jump		
1 110100 1 | disp-low | disp-high	直接 near jump		
1 111111 1 | mod 100 r/m	reg/mem 間接 near jump		
1 110101 0 | segment offset | segment selector	直接 far jump		リスト (c) 参照
1 111111 1 | mod 101 r/m (mod≠11)	間接 far jump		リスト (c) 参照

命令ニーモニックとコード	コメント	リアルモード	プロテクトモード
JE/JZ (jump on equal/zero) | 0 111010 0 | disp |	ZF=1 のときジャンプする.	○	○
JL/JNGE (jump on less/not greater or equal) | 0 111110 0 | disp |	(SF XOR OF)=1のときジャンプする.		
JLE/JNG (jump on less or equal/not greater) | 0 111111 0 | disp |	((SF XOR OF) OR ZF)=1 のときジャンプする.		
JB/JNAE (jump on below/not above or equal) | 0 111001 0 | disp |	CF=1 のときジャンプする.		
JBE/JNA (jump on below or equal/not above) | 0 111011 0 | disp |	(CF OR ZF)=1のときジャンプする.		
JP/JPE (jump on parity/parity even) | 0 111101 0 | disp |	PF=1 のときジャンプする.		
JO (jump on overflow) | 0 111000 0 | disp |	OF=1 のときジャンプする.		
JS (jump on sign) | 0 111100 0 | disp |	SF=1 のときジャンプする.		
JNE/JNZ (jump on not equal/not zero) | 0 111010 1 | disp |	ZF=0 のときジャンプする.		
JNL/JGE (jump on not less/greater or equal) | 0 111110 1 | disp |	(SF XOR OF)=0のときジャンプする.		
JNLE/JG (jump on not less or equal/greater) | 0 111111 1 | disp |	((SF XOR OF) OR ZF)=0 のときジャンプする.		
JNB/JAE (jump on not below/above or equal) | 0 111001 1 | disp |	CF=0 のときジャンプする.		
JNBE/JA (jump on not below or equal/above) | 0 111011 1 | disp |	(CF OR ZF)=0 のときジャンプする.		
JNP/JPO (jump on not parity/parity odd) | 0 111101 1 | disp |	PF=0 のときジャンプする.		
JNO (jump on not overflow) | 0 111000 1 | disp |	OF=0 のときジャンプする.		
JNS (jump on not sign) | 0 111100 1 | disp |	SF=1 のときジャンプする.		
LOOP (loop) | 1 110001 0 | disp |	CX←CX−1 CX=0 でなければジャンプする. (IP←IP+disp)		
LOOPZ/LOOPE (loop while zero/equal) | 1 110000 1 | disp |	CX←CX−1 CX≠0 かつ ZF=1 のときジャンプする.		
LOOPNZ/LOOPNE (loop while not zero/equal) | 1 110000 0 | disp |	CX←CX−1 CX≠0 かつ ZF=0 のときジャンプする.		
JCXZ (jump on CX zero) | 1 110001 1 | disp |	CX=0 のときジャンプする.		
ENTER (enter) | 1 100100 0 | data-low | data-high | L |		○	○
LEAVE (leave) | 1 100100 1 |		○	○

命令ニーモニックとコード	コ　メ　ン　ト	リアルモード	プロテクトモード
INT (interrupt) 1 100110 1 | type 1 100110 0	INT type INT 3	○	○ (リスト (f) 参照)
INTO (interrupt on overflow) 1 100111 0	OF=1のときタイプ4の割り込みに入る．	○	○ (リスト (f) 参照)
IRET (interrupt return) 1 100111 1	(NT=0) 割り込みリターン命令	○	○ (リスト (g) 参照)
	(NT=1) タスクスイッチ	×	○ (リスト (g) 参照)
BOUND (bound) 0 110001 0 | mod reg r/m		○	○
CLC (clear carry) 1 111100 0	CF←0	○	○
CMC (complement carry) 1 111010 1	(CF=0) CF←1 (CF=1) CF←0	○	○
STC (set carry) 1 111100 1	CF←1	○	○
CLD (clear direction) 1 111110 0	DF←0	○	○
STD (set direction) 1 111110 1	DF←1	○	○
CLI (clear interrupt) 1 111101 0	IF ←0	○	○ if IOPL ≧ CPL else GPerror (0)
STI (set interrupt) 1 111101 1	IF←1	○	○ if IOPL ≧ CPL else GPerror (0)
HLT (halt) 1 111010 0	ホルト命令 80286の動作を停止する．	○	○ if CPL=0 else GPerror (0)
WAIT (wait) 1 001101 1	$\overline{\text{BUSY}}$=Lowのとき，再び$\overline{\text{BUSH}}$=Highになるまで次の命令に進まない．	○	○
LOCK (lock prefix) 1 111000 0	LOCKの直後の命令実行におけるバスサイクルにおいて$\overline{\text{LOCK}}$=Lowとなる．	○	○
CLTS (clear task switch flag) 0 000111 1 | 0 000011 0	MSWのTS←0	○	○ if CPL=0 else GPerror (0)
ESC (80287 escape) 1 1011TT T | mod LLL r/m	80287の命令 TTT LLLのコードによって80287の各種の命令を表す．	○	○

命令ニーモニックとコード	コメント	リアルモード	プロテクトモード
SEG (segment override prefix) 0 01reg11 0	SEG の直後の命令において，メモリ参照に使用するセグメントレジスタを指定する．	○	○
LGDT (load GDTR) 0 000111 1 | 0 000000 1 | mod 010 r/m	GDTR ← mem	○	○ if CPL=0 else GPerror (0)
SGDT (store GDTR) 0 000111 1 | 0 000000 1 | mod 000 r/m	mem ← GDTR	○	○
LIDT (load IDTR) 0 000111 1 | 0 000000 1 | mod 011 r/m	IDTR ← mem	○	○ if CPL=0 else GPerror (0)
SIDT (store IDTR) 0 000111 1 | 0 000000 1 | mod 001 r/m	mem ← IDTR	○	○
LLDT (load LDTR) 0 000111 1 | 0 000000 0 | mod 010 r/m	LDTR ← reg/mem LDTR. CACHE ← GDT (LDTR)	×	○ if CPL=0 else GPerror (0)
SLDT (store LDTR) 0 000111 1 | 0 000000 0 | mod 000 r/m	reg/mem ← LDTR	×	○
LTR (load TR) 0 000111 1 | 0 000000 0 | mod 011 r/m	TR ← reg/mem TR. CACHE ← GDT (TR)	×	○ if CPL=0 else GPerror (0)
STR (store TR) 0 000111 1 | 0 000000 0 | mod 001 r/m	reg/mem ← TR	×	○
LMSW (load MSW) 0 000111 1 | 0 000000 1 | mod 110 r/m	MSW ← reg/mem	○	○ if CPL=0 else GPerror (0)
SMSW (load MSW) 0 000111 1 | 0 000000 1 | mod 100 r/m	reg/mem ← MSW	○	○
LAR (load access right) 0 000111 1 | 0 000001 0 | mod reg r/m	reg/mem で指定されるセレクタのディスクリプタのアクセスライトを reg の上位バイトに代入する．	×	○
LSL (load segment limit) 0 000111 1 | 0 000001 1 | mod reg r/m	reg/mem で指定されるセレクタのディスクリプタのリミットを reg に代入する．	×	○
ARPL (adjust requested privilege level) 0 000111 1 | 0 000001 1 | mod reg r/m	reg/mem で指定されるセレクタのディスクリプタのリミットを reg に代入する．	×	○
ARPL (adjust requested privilege level) 0 110001 1 | mod reg r/m		×	○
VERR (verify read access) 0 000111 1 | 0 000000 0 | mod 100 r/m	reg/mem で指定すれるセレクタのセグメントがリード可能かどうかを検査する．	×	○

VERW (verify write access) 0 000111 1 | 0 000000 0 | mod 101 r/m	reg/mem で指定されるセレクタのセグメントがライト可能かどうかを検査する.	×	○

表 付・2　mod のコードと意味

mod	意　　味
11	r/m はレジスタを表す.
00	disp を表すコードは 0 バイト
01	1 バイトの符号付きデータで disp を表す.
10	2 バイトの符号付きデータで disp を表す.

ただし, mod=00かつ r/m=110 のとき, 2 バイトの符号付きデータで disp を表す.

表 付・3　r/m のコードと意味 (mod=00, 01, 10 のとき r/m はメモリオペランドを表す)

r/m	EA (effective address)
000	[BX]+[SI]+disp
001	[BX]+[DI]+disp
010	[BP]+[SI]+disp
011	[BP]+[DI]+disp
100	[SI]+disp
101	[DI]+disp
110	[BP]+disp (ただし mod=00, r/m=110 のときのみ, EA=2 バイトの disp (直接指定) となる)
111	[BX]+disp

メモリ参照に実効的に使用されるオフセットを EA という.

表 付・4　r/m がレジスタを表すとき, また reg の識別

reg, r/m(mod=11)	レジスタ名	
	W=1	W=0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

表 付・5　reg がセグメントレジスタを表現するときのコード

sreg	セグメントレジスタ
00	ES
01	CS
10	SS
11	DS

リスト(a) DS, ES にセレクタ値 SEL を代入するとき，80286 は次の処理を実行する．

```
if SEL ≠ ヌルセレクタ
 then|if (SEL.INDEX+1)*8-1 ≦ DTR.LIMIT else GPerror(SEL);
      if (DT(SEL).S=0 AND (DT(SEL).E=0 OR DT(SEL).R=1))
       else GPerror(SEL);
      if DT(SEL).E=0 OR (DT(SEL).E=1 AND DT(SEL).C=0)
       then|if DT(SEL).DPL ≧ MAX(CS.CPL,SEL.RPL)
             else GPerror(SEL);
            if DT(SEL).P=1 else NPerror(SEL);|
       SREG ← SEL;  (* ここでSREGはDSまたはES *)
       SREG.CACHE ← DT(SEL);|
 else|(* セレクタがヌルセレクタの場合 *)
       SREG ← 0;  (* DS または ES に 0 を代入する *)
       SREG.CACHE ← invalid;(* キャッシュレジスタを 80286 が使用できないよ
                                うに不正であるという印を付ける *)
```

リスト(b) SS にセレクタ値 SEL を代入するとき，80286 は次の処理を実行する．

```
if SEL ≠ ヌルセレクタ else GPerror(0);
if (SEL.INDEX+1)*8-1 ≦ DTR.LIMIT else GPerror(SEL);
if (SEL.RPL=CS.CPL) else GPerror(SEL)
if DT(SEL).S=1 AND DT(SEL).E=0 AND DT(SEL).W=1
 else GPerror(SEL);
if DT(SEL).DPL=CS.CPL else GPerror(SEL);
if DT(SEL).P=1 else SSerror(SEL);
SS ← SEL;
SS.CACHE ← DT(SEL);
```

リスト(c) far JMP 命令を使用して CS の値を NEW_SEL に，IP を NEW_IP に変更するとき，80286 は次の処理を実行する．

```
if 間接 JMP
 then|if SREG.CACHE.E=0 OR (SREG.CACHE.E=1 AND SREG.
                                              CACHE.R=1)
        (* ここで，SREG は JMP 命令がメモリを参照するときに使用するセグメントレ
          ジスタを表す *)
      else GPerror(0);|
if NEW_SEL ≠ ヌル・セレクタ else GPerror(0);
if (NEW_SEL.INDEX+1)*8-1 ≦ DTR.LIMIT
 else GPerror(NEW_SEL);
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=1)
```

```
 then|(* NEW_SEL の指定するセグメントが特権例外のコードセグメントであるとき *)
        if DT(NEW_SEL).DPL ≦ CS.CPL else GPerror(NEW_SEL);
        if DT(NEW_SEL).P=1 else NPerror(NEW_SEL);
        if NEW_IP ≦ DT(NEW_SEL).LIMIT else GPerror(0);
        CS ← NEW_SEL;
        IP ← NEW_IP;
        CS.CACHE ← DT(NEW_SEL);|
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=0
 then|(* NEW_SEL の指定するセグメントがノンコンフォーミングコードセグメントである
          場合 *)
        if NEW_SEL.RPL ≦ CS.CPL else GPerror(NEW_SEL);
        if DT(NEW_SEL).DPL=CS.CPL else GPerror(NEW_SEL);
        if DT(NEW_SEL).P=1 else NPerror(NEW_SEL);
        if NEW_IP ≦ DT(NEW_SEL).LIMIT else GPerror(0);
        CS ← NEW_SEL;
        IP ← NEW_IP;
        CS.CACHE ← DT(NEW_SEL);
        CS.RPL ← SS.CPL;|
if DT(NEW_SEL)=CALL_GATE
 then|(* NEW_SEL が指定するディスクリプタがコールゲートである場合 *)
        if CALL_GATE.DPL ≧ CS.CPL else GPerror(NEW_SEL);
        if CALL_GATE.DPL ≧ NEW_SEL.RPL
         else GPerror(NEW_SEL);
        if CALL_GATE.P = 1 else NPerror(NEW_SEL);
        if CALL_GATE.SEL ≠ ヌルセレクタ else GPerror(0);
        if (CALL_GATE.SEL.INDEX+1)*8-1 ≦ DTR.LIMIT
         else GPerror(CALL_GATE.SEL);
        if DT(CALL_GATE.SEL).E=1
         else GPerror(CALL_GATE.SEL);
        if DT(CALL_GATE.SEL).C=0
         then|if DT(CALL_GATE.SEL).DPL = CS.CPL
               else GPerror(CALL_GATE.SEL);|
        if DT(CALL_GATE.SEL).C = 1
         then|if DT(CALL_GATE.SEL).DPL ≦ CS.CPL
               else GPerror(CALL_GATE.SEL);|
        if DT(CALL_GATE.SEL).P=1
         else NPerror(CALL_GATE.SEL);
        if CALL_GATE.IP ≦ DT(CALL_GATE.SEL).LIMIT
         else GPerror(0);
        CS ← CALL_GATE.SEL;
        IP ← CALL_GATE.IP;
        CS.CACHE ← DT(CS);
        CS.RPL ← SS.CPL;|
if DT(NEW_SEL) = TASK_GATE
 then|(* NEW_SEL が指定するディスクリプタがタスクゲートである場合 *)
```

```
        if TASK_GATE.DPL ≧ CS.CPL else GPerror(NEW_SEL);
        if TASK_GATE.DPL ≧ NEW_SEL.RPL
         else GPerror(NEW_SEL);
        if TASK_GATE.P = 1 else NPerror(NEW_SEL);
        if TASK_GATE.SEL.TI = 0
         else GPerror(TASK_GATE.SEL);
        if (TASK_GATE.SEL.INDEX+1)*8-1 ≦ DTR.LIMIT
         else GPerror(TASK_GATE.SEL);
        if DT(TASK_GATE.SEL).B = 0
         else GPerror(TASK_GATE.SEL);
        if DT(TASK_GATE.SEL).P = 1
         else NPerror(TASK_GATE.SEL);
        TASK_SWITCH;(* TSSセレクタがTASK_GATE.SELで指定されるタスクに
                       スイッチする *)
        if IP ≦ CS.CACHE.LIMIT else GPerror(0);|
if DT(NEW_SEL) = TSS_DESCRIPTOR
 then|(* NEW_SELが指定するディスクリプタがTSSディスクリプタである場合 *)
        if DT(NEW_SEL).DPL ≧ CS.CPL else GPerror(NEW_SEL);
        if DT(NEW_SEL).DPL ≧ NEW_SEL.RPL
         else GPerror(NEW_SEL);
        if DT(NEW_SEL).B = 0 else GPerror(NEW_SEL);
        if DT(NEW_SEL).P = 1 else NPerror(NEW_SEL);
        TASK_SWITCH;(* TSSセレクタがNEW_SELで指定されるタスクにスイッチす
                       る *)
        if IP ≦ CS.CACHE.LIMIT else GPerror(0);|
```

リスト(d)　far CALL命令を使用してCSの値をNEW_SELに，またIPの値をNEW_IPに変更するとき，80286は次の処理を実行する．

```
if 間接CALL
 then|if SREG.CACHE.E = 0 OR (SREG.CACHE.E=1
         AND SREG.CACHE.R=1)
         (* ここでSREGはCALL命令がメモリを参照するときに使用するセグメントレジ
            スタを表す *)
      else GPerror(0);|
if NEW_SEL ≠ ヌルセレクタ else GPerror(0);
if (NEW_SEL.INDEX+1)*8-1 ≦ DTR.LIMIT else GPerror
 else GPerror(NEW_SEL);
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=1
 then|(* NEW_SELの指定するセグメントが特権例外のコードセグメントである場合 *)
        if DT(NEW_SEL).DPL ≦ CS.CPL else GPerror(NEW_SEL);
        if DT(NEW_SEL).P = 1 else NPerror(NEW_SEL);
        if (SP-4) ≧ SS.CACHE.LIMIT (* スタックに戻りアドレスをPUSHす
                                      る空間があるかどうか *)
```

```
        else SSerror(0);
       if NEW_IP ≦ DT(NEW_SEL).LIMIT else GPerror(0);
       CS ← DT(NEW_SEL);
       CS ← NEW_SEL;
       IP ← NEW_IP;|
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=0
 then|(* NEW_SELの指定するセグメントがノンコンフォーミングコードセグメントである
         場合 *)
       if NEW_SEL.RPL ≦ CS.CPL else GPerror(NEW_SEL);
       if DT(NEW_SEL).DPL = CS.CPL else GPerror(NEW_SEL);
       if DT(NEW_SEL).P=1 else NPerror(NEW_SEL);
       if (SP-4) ≧ SS.CACHE.LIMIT (* スタックに戻りアドレスをPUSHす
                                    る空間があるかどうか *)
        else SSerror(0);
       if NEW_IP ≦ DT(NEW_SEL).LIMIT else GPerror(0);
       CS.CACHE ← DT(NEW_SEL);
       CS ← NEW_SEL;
       CS.RPL ← SS.CPL;
       IP ← NEW_IP;|
if DT(NEW_SEL) = CALL_GATE
 then|(* NEW_SELが指定するディスクリプタがコールゲートである場合 *)
       if CALL_GATE.DPL ≧ CS.CPL else GPerror(NEW_SEL);
       if CALL_GATE.DPL ≧ NEW_SEL.RPL
        else GPerror(NEW_SEL);
       if CALL_GATE.P = 1 else NPerror(NEW_SEL);
       if CALL_GATE.SEL ≠ ヌルセレクタ else GPerror(0);
       if (CALL_GATE.SEL.INDEX+1)*8-1 ≦ DTR.LIMIT
        else GPerror(CALL_GATE.SEL);
       if DT(CALL_GATE.SEL).E=1
        else GPerror(CALL_GATE.SEL);
       if DT(CALL_GATE.SEL).DPL ≦ CS.CPL
        else GPerror(CALL_GATE.SEL);
       if DT(CALL_GATE.SEL).C=0 AND DT(CALL_GATE.SEL).DPL
                                                      < CS.CPL
        then|(* より高い(数値的により小さい)特権レベルnのコードセグメントに制御を
                移行する場合 *)
             if TR.TSS.SSn  ≠ ヌルセレクタ (* TR.TSS.SSnはTRによ
                                             って指定される現在のTSS
                                             に記録された，特権レベルn
                                             で使用されるスタックセグメ
                                             ントのセグメントセレクタを
                                             表す *)
              else TSerror(0);
             if (TR.TSS.SSn.INDEX+1)*8-1 ≦ DTR.LIMIT
              else TSSerror(TR.TSS.SSn);
```

```
        if TR.TSS.SSn.RPL = DT(CALL_GATE.SEL).DPL
         else TSerror(TR.TSS.SSn);
        if DT(TR.TSS.SSn).DPL = DT(CALL_GATE.SEL).DPL
         else TSerror(TR.TSS.SSn);
        if DT(TR.TSS.SSn).E=0 AND DT(TR.TSS.SSn).W=1
         else TSerror(TR.TSS.SSn);
        if DT(TR.TSS.SSn).P=1
         else SSerror(TR.TSS.SSn);
        if TR.TSS.SPn-CALL_GATE.WC*2-8
         ≧ DT(TR.TSS.SSn).LIMIT+1
           (* 特権レベル n で使用するスタックに，以前のスタックポインタ，戻り
              アドレス，そしてコールゲートに指定されたワードのパラメータを記
              録する領域があるかどうか  *)
         else SSerror(0);
        if CALL_GATE.IP ≦ DT(CALL_GATE.SEL).LIMIT
         else GPerror(0);
        SS ← TR.TSS.SSn;
        SP ← TR.TSS.SPn;
        CS ← CALL_GATE.SEL;
        IP ← CALL_GATE.IP;
        CS.CACHE ← DT(CALL_GATE.SEL);
        SS.CACHE ← DT(TR.TSS.SSn);
        PUSH oldSS;(* 新しいスタックに以前のスタックの SS，SP を PUSH
                      する *)
        PUSH oldSP;
        (* count ワードだけ，以前のスタックから新しいスタックにパラメータを
           コピーする *)
        count ← CALL_GATE.WC AND 011111B;
        while(count=0)
        {PUSH oldSS:[oldSP+(count-1)*2];
         count ← count-1;}
        PUSH 戻り CS;
        PUSH 戻り IP;
        SS.CPL ← DT(SS).DPL;
        CS.RPL ← SS.CPL;}
   else{(* 特権レベルが変化しない場合 *)
        if SP-4 ≧ SS.CACHE.LIMIT+1 else SSerror(0);
        if CALL_GATE.IP ≦ DT(CALL_GATE.SEL).LIMIT
         else GPerror(0);
        CS ← CALL_GATE.SEL;
        IP ← CALL_GATE.IP;
        PUSH 戻り CS;
        PUSH 戻り IP;
        CS.CACHE ← DT(CALL_GATE.SEL);
        CS.RPL ← SS.CPL;}
```

```
if DT(NEW_SEL) = TASK_GATE
 then{(* NEW_SELが指定するディスクリプタがタスクゲートである場合 *)
     if TASK_GATE.DPL ≧ CS.CPL else GPerror(NEW_SEL);
     if TASK_GATE.DPL ≧ NEW_SEL.RPL
      else GPerror(NEW_SEL);
     if TASK_GATE.P = 1 else NPerror(NEW_SEL);
     if TASK_GATE.SEL.TI = 0(* TSSディスクリプタはGDTに定義しなけ
                               ればならない *)
      else GPerror(TASK_GATE.SEL);
     if (TASK_GATE.SEL.INDEX+1)*8-1 ≦ GDTR.LIMIT
      else GPerror(TASK_GATE.SEL);
     if DT(TASK_GATE.SEL).B = 0
      else GPerror(TASK_GATE.SEL);
     if DT(TASK_GATE.SEL).P = 1
      else NPerror(TASK_GATE.SEL);
     TASK_SWITCH;(* TSSセレクタがTASK_GATE.SELで指定されるタスクに
                   スイッチする *)
     if IP ≦ CS.CACHE.LIMIT else GPerror(0);}
if DT(NEW_SEL) = TSS_DESCRIPTOR
 then{(* NEW_SELが指定するディスクリプタがTSSディスクリプタである場合 *)
     if DT(NEW_SEL).DPL ≧ CS.CPL else GPerror(NEW_SEL);
     if DT(NEW_SEL).DPL ≧ NEW_SEL.RPL
      else GPerror(NEW_SEL);
     if DT(NEW_SEL).B = 0 else GPerror(NEW_SEL);
     if DT(NEW_SEL).P = 1 else NPerror(NEW_SEL);
     TASK_SWITCH;(* TSSセレクタがNEW_SELで指定されるタスクにスイッチす
                   る *)
     if IP ≦ CS.CACHE.LIMIT else GPerror(0);}
 else (* CALL命令が参照するセレクタが上のいずれでもない場合 *)
     GPerror(NEW_SEL);
```

リスト(e)　far RET 命令を実行するとき，80286は次のような処理を実行する．

```
if (SP+2) < 0FFFFH else SSerror(0);
if SS:[SP+2].RPL ≧ CS.CPL else GPerror(SS:[SP+2]);
if SS:[SP+2].RPL = CS.CPL
 then{(* 同じ特権レベルへのリターンの場合 *)
     if SS:[SP+2] ≠ ヌルセレクタ else GPerror(0);
     if (SS:[SP+2].INDEX+1)*8-1 ≦ DTR.LIMIT
      else GPerror(SS:[SP+2]);
     if DT(SS:[SP+2]).E = 1 else GPerror(SS:[SP+2]);
     if DT(SS:[SP+2]).C = 0
      then{(* ノンコンフォーミングコードセグメントの場合 *)
           if DT(SS:[SP+2]).DPL = CS.CPL
```

```
          else GPerror(SS:[SP+2]);|
    if DT(SS:[SP+2]).C=1
     then|(* コンフォーミング(特権例外)コードセグメントの場合 *)
          if DT(SS:[SP+2]).DPL ≦ CS.CPL
           else GPerror(SS:[SP+2]);|
    if DT(SS:[SP+2]).P=1 else NPerror(SS:[SP+2]);
    if SP > SS.CACHE.LIMIT+1 else SSerror(0);
    if SS:[SP] ≦ DT(SS:[SP+2]).LIMIT else GPerror(0);
    CS ← SS:[SP+2];
    IP ← SS:[SP];
    CS.CACHE ← DT(SS:[SP+2]);
    SP ← SP+4+imm;|(* imm はRET 命令のオペランドに指定した値を表すものと
                        する *)
else|(* より低い特権レベルへリターンする場合 *)
    if SP+8+imm ≦ OFFFFH (* 現在のスタックに戻りアドレスのポインタ，パラ
                              メータ，そして以前のスタックのポインタが記録
                              されているか *)
     else SSerror(0);
    if SS:[SP+2] ≠ ヌルセレクタ else GPerror(0);
    if (SS:[SP+2].INDEX+1)*8-1 ≦ DTR.LIMIT
     else GPerror(SS:[SP+2]);
    if DT(SS:[SP+2]).E=1 else GPerror(SS:[SP+2]);
    if DT(SS:[SP+2]).C=0
     then|(* ノンコンフォーミングコードセグメントの場合 *)
          if DT(SS:[SP+2]).DPL = SS:[SP+2].RPL
           else GPerror(SS:[SP+2]);|
    if DT(SS:[SP+2]).C=1
     then|(* コンフォーミング(特権例外)コードセグメントの場合 *)
          if DT(SS:[SP+2]).DPL ≦ SS:[SP+2].RPL
           else GPerror(SS:[SP+2]);|
    if DT(SS:[SP+2]).P = 1 else NPerror(SS:[SP+2]);
    if SS:[SP+6+imm] ≠ ヌルセレクタ
       (* SS:[SP+6+imm]には以前のスタックの SS のセレクタが記録されてい
          る *)
     else GPerror(0);
    if (SS:[SP+6+imm].INDEX+1)*8-1 ≦ DTR.LIMIT
     else GPerror(SS:[SP+6+imm]);
    if SS:[SP+6+imm].RPL = SS:[SP+2].RPL
     else GPerror(SS:[SP+6+imm]);
    if DT(SS:[SP+6+imm]).E = 0
       AND DT(SS:[SP+6+imm]).W=1
     else GPerror(SS:[SP+6+imm]);
    if DT(SS:[SP+6+imm]).DPL = SS:[SP+2].RPL
     else GPerror(SS:[SP+6+imm]);
    if DT(SS:[SP+6+imm]).P = 1
```

```
   else NPerror(SS:[SP+6+imm]);
  if SS:[SP] ≦ DT(SS:[SP+2]).LIMIT+1
   else GPerror(0);
  SS.CPL ← SS:[SP+2].RPL;
  CS ← SS:[SP+2];
  IP ← SS:[SP];
  CS.RPL ← SS.CPL;
  SP ← SP+4+imm;
  SS ← SS:[SP+2];
  SP ← SS:[SP];
  SP ← SP+imm;
  CS.CACHE ← DT(CS);
  SS.CACHE ← DT(SS);
  (* DS, DS.CACHE に不正な値が入っていないか検査する．もしリターンした後の状
     況に合わないような値をもつときはDSにヌルセレクタを代入し，DS.CACHEを使
     用できないようにする *)
  if (DS.INDEX+1)*8-1 ≦ DTR.LIMIT
   else{DS ← 0;  (* DSにヌルセレクタ0を代入し，DS.CACHEを使用でき
                        ないようにする *)
        DS.CACHE ← invalid;}
  if DT(DS).E=0 OR (DT(DS).E=1 AND DT(DS).R=1)
   else{DS ← 0;DS.CACHE ← invalid;}
  if DT(DS).E=0 OR (DT(DS).E=1 AND DT(DS).C=0)
   then{if DT(DS).DPL ≧ MAX(CS.CPL,DS.RPL)
         else{DS ← 0;DS.CACHE ← invalid;}
  (* ES についても，DS の場合と全く同様の処理を実行する *)
```

リスト(f)　割り込みタイプINT_VECの割り込みにおいて，80286は次の処理を実行する．

```
if (INT_VEC+1)*8-1 ≦ IDTR.LIMIT
 else GPerror(INT_VEC*8+2+EXT)
if (IDT(INT_VEC) = INT_GATE) OR (IDT(INT_VEC) = TRAP_GATE
   OR (IDT(INT_VEC) = TASK_GATE
   (* INT_VEC の指定するゲートが割り込みゲート，トラップゲート，タスクゲートのいずれ
      かである *)
 else GPerror(INT_VEC*8+2+EXT);
if INTinstruction(* INT 命令などのソフトウェア割り込みによって割り込み処理を実
                    行する場合 *)
 then{if IDT(INT_VEC).DPL ≧ CS.CPL
       else GPerror(INT_VEC*8+2+EXT);}
if IDT(INT_VEC).P=1 else NPerror(INT_VEC*8+2+EXT);
if (IDT(INT_VEC) = INT_GATE) OR (IDT(INT_VEC)=TRAP_GATE)
```

```
then|(* 割り込みゲートまたはトラップゲートの場合 *)
      if IDT(INT_VEC).SEL ≠ ヌルセレクタ else GPerror(EXT);
      if (IDT(INT_VEC).SEL.INDEX+1)*8-1 ≦ DTR.LIMIT
       else GPerror(IDT(INT_VEC).SEL+EXT);
      if DT(IDT(INT_VEC).SEL).E = 1
       else GPerror(IDT(INT_VEC).SEL+EXT);
      if DT(INT_VEC).SEL).P = 1
       else NPerror(IDT(INT_VEC).SEL+EXT);
      if DT(IDT(INT_VEC).SEL).C = 0
         AND (DT(IDT(INT_VEC).SEL).DPL < CS.CPL)
       then|(* ノンコンフォーミングコードセグメントで，より高い特権レベルに移行する
              場合 *)
            if TR.TSS.SSn ≠ ヌルセレクタ else GPerror(EXT);
            if (TR.TSS.SSn.INDEX+1)*8-1 < DTR.LIMIT
             else TSerror(SSn+EXT);
            if TR.TSS.SSn.RPL = DT(IDT(INT_VEC).SEL).DPL
             else TSerror(SSn+EXT);
            if DT(TR.TSS.SSn).DPL = DT(IDT(INT_VEC).SEL).
               DPL
             else TSerror(SSn+EXT);
            if (DT(TR.TSS.SSn).E=0)
               AND (DT(TR.TSS.SSn).W=1)
             else TSerror(SSn+EXT);
            if DT(TR.TSS.SSn).P = 1
             else SSerror(SSn+EXT);
            if TR.TSS.SPn-10 ≧ DT(TR.TSS.SSn).LIMIT+1
             else SSerror(0);
            if IDT(INT_VEC).IP ≦
               DT(IDT(INT_VEC).SEL).LIMIT
             else GPerror(0);
            SS ← TR.TSS.SSn;
            SP ← TR.TSS.SPn;
            CS ← IDT(INT_VEC).SEL;
            IP ← IDT(INT_VEC).IP;
            CS.CACHE ← DT(CS);
            SS.CACHE ← DT(SS);
            PUSH oldSS;
            PUSH oldSP;
            PUSH flag register;
            PUSH 戻り CS;
            PUSH 戻り IP;
            SS.CPL ← DT(IDT(INT_VEC).SEL).DPL;
            CS.RPL ← SS.CPL
            if error_code then(* エラーコードがある場合 *)
               PUSH error_code;
```

```
        if IDT(INT_VEC) = INT_GATE
         then (* 割り込みゲートの場合 *) FLAG.IF ← 0;
        FLAG.TF ← 0;
        FLAG.NT ← 0;|
  if (DT(IDT(INT_VEC).SEL).C=1) OR (DT(IDT(INT_VEC).
     SEL).DPL = CS.CPL)
   then|(* コンフォーミングコードセグメントか特権レベルが変化しない場合
           *)
        if (SP-6) ≧ SS.CACHE.LIMIT+1
         else SSerror(0);
        if error_code
         then|(* エラーコードがある場合 *)
              if (SP-8) ≧ SS.CACHE.LIMIT+1
               else SSerror(0);|
        if IDT(INT_VEC).IP ≦
           DT(IDT(INT_VEC).SEL).LIMIT
         else GPerror(0);
        PUSH flag register;
        PUSH 戻り CS;
        PUSH 戻り IP;
        CS ← IDT(INT_VEC).SEL;
        IP ← IDT(INT_VEC).IP;
        CS.CACHE ← DT(IDT(INT_VEC).SEL);
        CS.RPL ← SS.CPL;
        if error_code then (* エラーコードがある場合 *)
           PUSH error_code;
        if IDT(INT_VEC) = INT_GATE
         then (* 割り込みゲートの場合 *) FLAG.IF ← 0;
        FLAG.TF ← 0;
        FLAG.NP ← 0;|
   else GPerror(IDT(INT_VEC).SEL+EXT);
  if IDT(INT_VEC) = TASK_GATE
   then|(* 割り込みによってタスクゲートを参照した場合 *)
        if TASK_GATE.SEL.TI = 0
           (* TSS ディスクリプタは GDT に定義しなければならない *)
         else GPerror(TASK_GATE.SEL);
        if (TASK_GATE.SEL.INDEX+1)*8-1 ≦ GDTR.LIMIT
         else GPerror(TASK_GATE.SEL);
        if DT(TASK_GATE.SEL).B = 0
         else GPerror(TASK_GATE.SEL);
        if DT(TASK_GATE.SEL).P = 1
         else NPerror(TASK_GATE.SEL);
        TASK_SWITCH;(* TSS セレクタが TASK_GATE.SEL で指定され
                      る新しいタスクにスイッチする *)
        if error_code
```

```
                    then|(* エラーコードがある場合 *)
                         if SP-2 ≧ SS.CACHE.LIMIT+1
                          else SSerror(0);
                         PUSH error_code;|
                   if IP ≦ CS.CACHE.LIMIT else GPerror(0);|
```

リスト(g) IRET命令において，80286は次の処理を実行する

```
if FLAG.NT = 1
 then|(* タスクスイッチとして動作する *)
      if TR.TSS.BACK_LINK.TI = 0
       else TSerror(TR.TSS.BACK_LINK);
      if (TR.TSS.BACK_LINK.INDEX+1)*8-1 ≦ GDTR.LIMIT
       else TSerror(TS.TSS.BACK_LINK);
      if DT(TR.TSS.BACK_LINK).AR = TSS_DESCRIPTOR
         (* TSSのバックリンクに記録されたセレクタがTSSディスクリプタを指定す
            る *)
       else TSerror(TR.TSS.BACK_LINK);
      if DT(TR.TSS.BACK_LINK).B=1
       else TSerror(TR.TSS.BACK_LINK);
      if DT(TR.TSS.BACK_LINK).P=1
       else TSerror(TR.TSS.BACK_LINK);
      TASK_SWITCH;(* BACK_LINKで指定されるTSSセレクタの新しいタスクに
                     スイッチする *)
      if IP ≦ CS.CACHE.LIMIT else GPerror(0);|
if FLAG.NT = 0
 then|(* 割り込みからのリターン命令として動作する *)
      if (SP+2) < OFFFFH else SSerror(0);
      if SS:[SP+2].RPL ≧ CS.CPL
       else GPerror(SS:[SP+2]);
      if SS:[SP+2].RPL = CS.CPL
       then|(* 同じ特権レベルのセグメントに制御を移行する場合 *)
            if (SP+4) < OFFFFH else SSerror(0);
            if SS:[SP+2] ≠ ヌルセレクタ else GPerror(0);
            if (SS:[SP+2].INDEX+1)*8-1 ≦ DTR.LIMIT
             else GPerror(SS:[SP+2]);
            if DT(SS:[SP+2]).E = 1
             else GPerror(SS:[SP+2]);
            if DT(SS:[SP+2]).C = 0
             then|(* ノンコンフォーミングコードセグメントの場合 *)
                  if DT(SS:[SP+2]).DPL = CS.CPL
                   else GPerror(SS:[SP+2]);|
             else|(* コンフォーミングコードセグメントの場合 *)
                  if DT(SS:[SP+2]).DPL ≦ CS.CPL
```

```
        else GPerror(SS:[SP+2]);|
  if DT(SS:[SP+2]).P=1
   else NPerror(SS:[SP+2]);
  if SS:[SP] ≦ DT(SS:[SP+2]).LIMIT
   else GPerror(0);
  CS ← SS:[SP+2];
  IP ← SS:[SP];
  CS.CACHE ← DT(CS);
  FLAG ← SS:[SP+4];
  SP ← SP+6;|
else|(* より低い特権レベルにリターンする場合 *)
  if (SP+8) < 0FFFFH else SSerror(0);
  if SS:[SP+2] ≠ ヌルセレクタ
   else GPerror(0);
  if (SS:[SP+2].INDEX+1)*8-1 ≦ DTR.LIMIT
   else GPerror(SS:[SP+2]);
  if DT(SS:[SP+2]).E = 1
   else GPerror(SS:[SP+2]);
  if DT(SS:[SP+2]).C = 0
   then|(* ノンコンフォーミングコードセグメントの場合 *)
  if DT(SS:[SP+2]).DPL = CS.CPL
   else GPerror(SS:[SP+2]);|
   else|(* コンフォーミング（特権例外）コードセグメントの場
        合 *)
       if DT(SS:[SP+2]).DPL > CS.CPL
        else GPerror(SS:[SP+2]);|
  if DT(SS:[SP+2]).P = 1
   else NPerror(SS:[SP+2]);
  if SS:[SP+8] ≠ ヌルセレクタ else GPerror(0);
  if (SS:[SP+8].INDEX+1)*8-1 ≦ DTR.LIMIT
   else GPerror(SS:[SP+8]);
  if SS:[SP+8].RPL = SS:[SP+2].RPL
   else GPerror(SS:[SP+8]);
  if DT(SS:[SP+8]).E = 0
     AND DT(SS:[SP+8]).W = 1
   else GPerror(SS:[SP+8]);
  if DT(SS:[SP+8]).DPL = SS:[SP+2].RPL
   else GPerror(SS:[SP+8]);
  if DT(SS:[SP+8]).P = 1
   else NPerror(SS:[SP+8]);
  if SS:[SP] ≦ DT(SS:[SP+2]).LIMIT
   else GPerror(0);
  CS ← SS:[SP+2];
  IP ← SS:[SP];
  FLAG ← SS:[SP+4];
```

```
SS ← SS:[SP+8];
SP ← SS:[SP+6];
CS.RPL ← SS:[SP+2].RPL;
CS.CACHE ← DT(CS);
SS.CACHE ← DT(SS);
(* DS, DS.CACHE に不正な値が入っていないか検査する．も
   し，リターンした後の状況に合わないような値をもつときは DS
   にヌルセレクタを代入し，DS.CACHE を使用できないよう
   にする *)
if (DS.INDEX+1)*8-1 ≦ DTR.LIMIT
 else|DS ← 0;DS.CACHE ← invalid;|
if DT(DS).E = 0 OR (DT(DS).E = 1
   AND DT(DS).R = 1)
 else|DS ← 0;DS.CACHE ← invalid;|
if DT(DS).E = 0 OR (DT(DS).E = 1
   AND DT(DS).C = 0)
 then|if DT(DS).DPL ≧ MAX(CS.CPL,DS.RPL)
        else|DS ← 0;DS.CACHE ← invalid;||
(* ES についても DS の場合と全く同様の処理を実行する *)
```

リスト(h) TASK_SWITCH TSS セレクタ NEWTSS_SEL で指定される新しいタスクにスイッチするとき 80286 は次のような手順を実行する．

```
DT(NEWTSS_SEL).B ← 1;
if TR.CACHE.LIMIT ≧ 43 else TSerror(NEWTSS_SEL);
if DT(NEWTSS_SEL).LIMIT ≧ 43 else TSerror(NEWTSS_SEL);
TR.TSS ← (AX,BX,CX,DX,SI,DI,BP,SP,IP,FLAG,CS,DS,ES,SS,
         LDTR);
         (* 現在のレジスタの状態を現在の TSS に記録する *)
if CALL or interrupt cause task switch
 then (* CALL 命令または割り込みによって実行されるタスクスイッチの場合 *)
      NEWTSS_SEL.TSS.BACK_LINK ← TR;(* 新しいタスクの TSS のバック
                                       リンクに現在のタスクの TSS セ
                                       レクタを記録する *)
DT(TR).B ← 0;
TR ← NEWTSS_SEL;
TR.CACHE ← DT(TR);
(AX,BX,CX,DX,SI,DI,BP,SP,IP,FLAG,CS,DS,ES,SS,LDTR)
 ← TR.TSS;
 (* 新しい TSS に記録されていたレジスタの状態を 80286 のレジスタに代入する *)
(* LDTR キャッシュとセグメントキャッシュは新しい値を代入するまで使用できないようにす
   る *)
LDTR.CACHE ← invalid;
SS.CACHE ← invalid;
```

```
CS.CACHE ← invalid;
DS.CACHE ← invalid;
ES.CACHE ← invalid;
MSW.TS ← 1;
if CALL or interrupt cause task switch
 then (* CALL 命令，割り込みによって実行されるタスクスイッチの場合 *)
      FLAG.NT ← 1;
if (TR.TSS.LDT_SEL.INDEX+1)*8-1 ≦ GDTR.LIMIT
   (* ここで，TR.TSS.LDT_SEL は現在の TSS に記録された LDT セレクタを表す *)
 else TSerror(TR.TSS.LDT_SEL);
if DT(TR.TSS.LDT_SEL) = LDT_DESCRIPTOR
 else TSerror(TR.TSS.LDT_SEL);
if DT(TR.TSS.LDT_SEL).P = 1 else TSerror(TR.TSS.LDT_SEL);
LDTR.CACHE ← DT(LDTR);
CS.CPL ← TR.TSS.CS.RPL
if SS ≠ ヌルセレクタ else TSerror(SS);
if (SS.INDEX+1)*8-1 ≦ DTR.LIMIT else TSerror(SS);
if SS.RPL = CS.CPL else TSerror(SS);
if DT(SS).DPL = CS.CPL else TSerror(SS);
if DT(SS).E = 0 AND DT(SS).W = 1 else TSerror(SS);
if DT(SS).P = 1 else SSerror(SS);
SS.CACHE ← DT(SS);
if CS ≠ ヌルセレクタ else TSerror(CS);
if (CS.INDEX+1)*8-1 ≦ DTR.LIMIT else TSerror(CS);
if DT(CS).E = 1 else TSerror(CS);
if DT(CS).C = 0
 then|(* ノンコンフォーミングコードセグメントの場合 *)
      if DT(CS).DPL = CS.CPL else TSerror(CS);|
 else|(* コンフォーミングコードセグメントの場合 *)
      if DT(CS).DPL ≦ CS.CPL else TSerror(CS);|
if DT(CS).P = 1 else NPerror(CS);
CS.CACHE ← DT(CS);
(* DS キャッシュの決定 *)
if DS ≠ ヌルセレクタ
 then|if (DS.INDEX+1)*8-1 ≦ DTR.LIMIT else TSerror(DS);
      if DT(DS).E = 0 OR (DT(DS).E = 1 AND DT(DS).R=1)
       else TSerror(DS);
      if DT(DS).E = 0 OR (DT(DS).E = 1 AND DT(DS).R = 1
         AND DT(ES).C = 0)
       then|if DT(DS).DPL ≧ CS.CPL else TSerror(DS);
            if DT(DS).DPL ≧ DS.RPL else TSerror(DS);|
      if DT(DS).P = 1 else NPerror(DS);
      DS.CACHE ← DT(DS);|
(* ESキャッシュの場合も上のDSキャッシュとまったく同様に決まる *)
```

参 考 文 献

(1) iAPX 286 プログラマーズリファレンスマニュアル

(2) iAPX 286 ハードウェアリファレンスマニュアル

(3) iAPX 286 オペレーティングシステムライターズガイド

(4) iSBC 286/10 A single board computer hardware reference manual

おわりに

インテル社製品のよいところはマイクロプロセッサのチップとともに，すぐれた開発環境と iRMX，XENIX などのようなソフトウェア環境が提供されるところにもある．逆にいえば，インテル社の提供するアセンブリ言語，PL/M 言語などの高級言語，各種のユーティリティソフト，そして iRMX などの OS などを含めてのソフトウェア環境を知らずに，チップのアーキテクチャだけを勉強した場合，そのマイクロプロセッサを誤解する可能性がある．

たとえば，8086，80286 はセグメンテーションを採用しているからプログラムが困難であるという批判を耳にすることがあるが，それはおそらくセグメントを扱う方法が悪いために，プログラミングにおいて問題が発生しているのだろう．インテルのアセンブリ言語（ASM 86，ASM 286 など）がもついくつかの疑似命令は，セグメントをもつマイクロプロセッサのプログラミング上の困難さを解決する．なお，セグメント自体はミニコン以上のレベルの汎用計算機のアーキテクチャとして提案され，使用されてきた考え方であり，もし 8086 のセグメントを批判するとすれば，それはセグメント自体にではなく，セグメントの最大の大きさが 64 K バイトの制限をもつことに対してであろう．しかし，このことも 8086 が開発された時代は，そのときのメモリチップの容量を考えれば，グラフィックワークステーションなどのように大量のリニアなメモリ空間を必要とする応用にマイクロプロセッサを利用することは考えられなかった時期であり，無理のないことである．80286 は 8086 のソフトウェアをマルチタスク環境で実行するということが第一義にあるから，セグメントの最大の大きさは 64 K バイトであるが，80386 のセグメントの最大の大きさは 4 G バイトになっている．また，80286 の GDT，LDT，IDT，またそれらのテーブルで定義するディスクリプタ，そして TSS を扱う困難さはビルダ BLD 286 が解決する．

索　　　　引

タ　行

ワ　　行

アルファベット

著 者 略 歴

昭和53年　近畿大学理工学部電子工学科卒業
インテルジャパン株式会社を経て
現　在　日本カルコンプ株式会社

図解 16ビットマイクロコンピュータ 80286 の 使 い 方

1987年 5 月20日　第1版第1刷発行
1989年 5 月20日　第1版第4刷発行

OHM・OHM・OHM
著者承認
検印省略

著　者　本(もと) 岡(おか) 善(よし) 剛(たけ)

発行者　株式会社 オ ー ム 社
代表者 種 田 則 一

発行所　株式会社 オ ー ム 社
郵便番号 101
東京都千代田区神田錦町 3-1
振 替　東京 6 - 20018
電 話　03(233)0641(代表)

印刷　中央印刷　製本　関川製本所

Printed in Japan

ISBN 4 - 274 - 07349 - 1

書名	著者	判型
マイクロコンピュータハンドブック	渡辺・正田・矢田共編	B5判
オンラインシステムの設計	甘利直幸著	A5判
プログラマ適性検査	三重野重三著	A5判
入門ソフトウェア	高橋守清著	A5判
JISに準拠した FORTRAN（演習コース）	大泉充郎監修	A5判
BASICによる システム指向プログラミング	永村文孝著	B5判
マイクロコンピュータ入門テキスト	湯田・伊藤共著	A5判
絵ときマイコン実習	伊藤・橋本共著	A5判
続制御用マイコンの作り方・使い方	北川一雄著	B5判
マイコンロボットの設計と製作	堀・菅谷共編	B5判
FMシリーズ機械語入門	脇 英世著	A5判
Z－80機械語によるプログラムと制御	中山 章著	A5判
図解 16ビットマイクロコンピュータ 80286の使い方	本岡善剛著	A5判
図解 6800／6809による マイコンアセンブラ入門	桐山 清著	B5判
図解マイクロコンピュータ 8086アセンブラプログラミング入門	井出裕巳著	B5判
絵とき コンカレントCP／M入門	荒武達男編	B5判
IBM 5550 活用法	鈴木智彦編著	B5判
IBM 5550とJX BASIC入門	鈴木 昇編	B5判
FACOM 9450シリーズ BASIC入門	鈴木 昇編著	B5判
わかるCAD／CAM	ばんざい秀男著	B5判
パソコンネットワーク入門	松山佐和著	B5判
パソコンマーケティング入門	阿部将美著	B5判
パーソナルコンピュータのための OS－9入門	蜂谷 博著	A5判
パソコンデータベース活用法	酒井・葛井・阿部共著	B5判
パソコンによる パーソナルプロッタ・プログラミング	杉浦義人著	B5判
パソコンによる 透視図の作り方	岡本 博著	B5判
家庭で役立つ パソコン入門	岡田千恵子著	B5判
親子で学ぶ パソコン教室	岩佐澄男監修	B5判
小学生のための パソコン教室	松山佐和・松山優治共著	B5判
中学生のための パソコン教室	吉田・木村共著	B5判
中学生のための BASIC教室	吉田・木村共著	B5判
高校生のための パソコン教室	竹内・目七共著	B5判

カバー印刷：池田印刷